O'REILLY®

高性能MySQL（第3版）

High Performance MySQL, Third Edition

Baron Schwartz, Peter Zaitsev, Vadim Tkachenko 著

宁海元 周振兴 彭立勋 翟卫祥 刘辉 译

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

VISIT…

LANZAROTE Caliente.COM

内 容 简 介

本书是 MySQL 领域的经典之作，拥有广泛的影响力。第 3 版更新了大量的内容，不但涵盖了最新 MySQL 5.5 版本的新特性，也讲述了关于固态盘、高可扩展性设计和云计算环境下的数据库相关的新内容，原有的基准测试和性能优化部分也做了大量的扩展和补充。全书共分为 16 章和 6 个附录，内容涵盖 MySQL 架构和历史，基准测试和性能剖析，数据库软硬件性能优化，复制、备份和恢复，高可用与高可扩展性，以及云端的 MySQL 和 MySQL 相关工具等方面的内容。每一章都是相对独立的主题，读者可以有选择性地单独阅读。

本书不但适合数据库管理员（DBA）阅读，也适合开发人员参考学习。不管是数据库新手还是专家，相信都能从本书有所收获。

©2012 by Baron Schwartz, Peter Zaitsev, Vadim Tkachenko.
Simplified Chinese Edition, jointly published by O'Reilly Media,Inc. and Publishing House of Electronics Industry, 2013.
Authorized translation of the English edition, 2012 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.
All rights reserved including the rights of reproduction in whole or in part in any form.

本书简体中文版专有出版权由 O'Reilly Media,Inc.授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号　图字：01-2013-1661

图书在版编目（CIP）数据

高性能 MySQL：第 3 版 /（美）施瓦茨（Schwartz,B.），（美）扎伊采夫（Zaitsev,P.），（美）特卡琴科（Tkachenko,V.）著；宁海元等译. —北京：电子工业出版社，2013.5
书名原文：High Performance MySQL, Third Edition
ISBN 978-7-121-19885-4

Ⅰ. ①高… Ⅱ. ①施… ②扎… ③特… ④宁… Ⅲ.①关系数据库系统 Ⅳ. ①TP311.138

中国版本图书馆 CIP 数据核字（2013）第 054420 号

策划编辑：张春雨
责任编辑：白　涛　　贾　莉
封面设计：Karen Montgomery　张　健
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：50　字数：1040 千字
印　　次：2013 年 5 月第 1 次印刷
定　　价：128.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

O’Reilly Media, Inc.介绍

O’Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O’Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项O’Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar博客有口皆碑。”

——Wired

“O’Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O’Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

译者序

在互联网行业，MySQL 数据库毫无疑问已经是最常用的数据库。LAMP（Linux + Apache + MySQL + PHP）甚至已经成为专有名词，也是很多中小网站建站的首选技术架构。我所在的公司淘宝网，在 2003 年非典肆虐期间创立时，选择的就是 LAMP 架构，当时 MySQL 的版本还是 4.0。但是到了 2003 年底，由于业务超预期的增长，MySQL 4.0（当时用的还是 MyISAM 引擎）的很多缺点在高并发大压力下暴露了出来，于是技术上开始改用商业的 Oracle 数据库。随后几年 Oracle 加小型机和高端存储的数据库架构支撑了淘宝网业务的爆炸式增长，数据库也从最初的两三个库增长到十几个库，并且每个库的硬件已经逐步升级到顶配，“天花板”很明显地摆在了眼前。于是在 2008 年，基于 PC 服务器的 MySQL 数据库再次成为 DBA 团队的选择，这时候 MySQL 的稳定版本已经升级到 5.0，并且 5.1 也已经在开发中，性能和特性相对于 2003 年的时候已经有了非常大的提升。淘宝网的数据库架构也逐渐从垂直拆分走向水平拆分，在大规模水平集群的架构设计中，开源的 MySQL 受到的关注度越来越高，并且一年多来的实践也证明了 MySQL（存储引擎主要使用的是 InnoDB）在高压力下的可用性。于是从 2009 年开始，后来颇受外界关注的所谓“去 IOE”开始实施，经过三年多的架构改造，到 2012 年整个淘宝网的核心交易系统已经全部运行在基于 PC 服务器的 MySQL 数据库集群中，全部实例数超过 2000 个。今年的“双 11”大促中，MySQL 单库经受了最高达 6.5 万的 QPS，某个拥有 32 个节点的核心集群的总 QPS 则稳定在 86 万以上，并且在整个大促（包括之前三年的“双 11”大促）期间，数据库未发生过任何影响大促的重大故障。当然，这个结果，也得益于淘宝网整个应用架构的设计，以及这几年来革命性的闪存设备的迅猛发展。

2008 年，淘宝 DBA 团队准备从 Oracle 转向 MySQL 的时候，团队中的大多数人对 MySQL 的了解都非常之少。当时国内技术圈对 MySQL 的讨论也不多见，网上能找到的大多数中文资料基本上关注的还是如何安装，如何配置主备复制等。而 MySQL 中文

类的书籍，大部分还是和 PHP 放在一起，作为 PHP 开发中的一环来讲述的。所以当我们发现 mysqlperformanceblog.com 这个相当专业的国外博客的时候，无不欣喜莫名。同时也知道了博客的作者们 2008 年出版的 *High Performance MySQL* 第二版（中文版于 2010 年 1 月出版），这本书被很多 MySQL DBA 们奉为圭臬，书的三位主要作者 Baron Schwartz、Peter Zaitsev 和 Vadim Tkachenko 也在 MySQL DBA 圈中耳熟能详，他们组建的 Percona 公司和 Percona Server 分支版本以及 XtraDB 存储引擎也逐渐为国内 DBA 所熟知。2011 年 12 月，淘宝网和 O'Reilly 在北京联合举办的 Velocity China 2011 技术大会上，我们有幸邀请到 Percona 公司的华人专家季海东（目前已离职）来介绍 MySQL 5.5 InnoDB/XtraDB 的性能优化和诊断方法。在季海东先生的引荐下，我们也和 Peter 通过 Skype 电话会议有过沟通，介绍了 MySQL 在淘宝的应用情况，我们对 MySQL 一些特性的需求，以及对 MySQL 做的一些 patch，并随后保持了密切的邮件联系。有了这些铺垫，我们对于在生产系统中采用 Percona Server 5.5 也有了更大的信心，如今已有超过 1000 个 Percona Server 5.5 的实例在线上运行。所以今年上半年电子工业出版社的张春雨（侠少）编辑找到我来翻译本书的第三版的时候，很是激动，一口应承。

考虑到这么经典的书应该尽快地和读者见面，故此我邀请了团队中的 MySQL 专家周振兴（花名：苏普）、彭立勋、翟卫祥（花名：印风）、刘辉（花名：希羽）一起来翻译。其中，我负责前、推荐序和第 1、2、3 章，周振兴负责第 5、6、7 章，彭立勋负责第 4、8、9、14 章，翟卫祥负责第 10、11、12、13 章，刘辉负责第 15、16 章和附录部分，最后由我负责统稿。所以毫无疑问，这本书是团队合作的结晶。虽然我们满怀激情，但由于都是第一次参与翻译技术书籍，确实对困难有些预估不足，加上下半年为了准备“双 11”等各种大促，需要在 DBA 团队满负荷的工作间隙挤出个人时间，初稿出来后，由于每个人翻译风格不太一致，几次审稿修订，也让本书的编辑李云静和白涛吃了不少苦头，在此对大家表示深深的感谢，是大家不懈的努力，才使得本书能够顺利地和读者见面。但书中肯定还存在不少问题，恳请读者不吝指出，欢迎大家和我的新浪微博 *http://weibo.com/NinGoo* 进行互动。

同时还要感谢本书第二版的译者们，他们娴熟的语言技巧给了我们很多的参考。也要感谢帮助审稿的同事们，包括但并不仅限于张新铭（花名：俊达）、张瑞（花名：张瑞）、吴学章（花名：维西）等，彭立勋甚至还发动了他女朋友加入到审稿工作中，在此一并表示感谢。当然，最后还要感谢我的妻子 Lalla，在我占用了大量周末时间的时候能够给予支持，并承担了全部的家务，让我以译书为借口毫无心理负担地偷懒。

宁海元（花名：江枫）

2013 年 3 月 于余杭

目录

推荐序 xxiii

前言 xxv

第 1 章 MySQL 架构与历史 1

1.1 MySQL 逻辑架构 1
1.1.1 连接管理与安全性 2
1.1.2 优化与执行 3
1.2 并发控制 3
1.2.1 读写锁 4
1.2.2 锁粒度 4
1.3 事务 6
1.3.1 隔离级别 8
1.3.2 死锁 9
1.3.3 事务日志 10
1.3.4 MySQL 中的事务 10
1.4 多版本并发控制 12
1.5 MySQL 的存储引擎 13
1.5.1 InnoDB 存储引擎 16
1.5.2 MyISAM 存储引擎 17
1.5.3 MySQL 内建的其他存储引擎 19

1.5.4 第三方存储引擎 22
1.5.5 选择合适的引擎 24
1.5.6 转换表的引擎 27
1.6 MySQL 时间线（Timeline） 29
1.7 MySQL 的开发模式 32
1.8 总结 33

第 2 章 MySQL 基准测试 35

2.1 为什么需要基准测试 35
2.2 基准测试的策略 37
2.2.1 测试何种指标 38
2.3 基准测试方法 40
2.3.1 设计和规划基准测试 41
2.3.2 基准测试应该运行多长时间 42
2.3.3 获取系统性能和状态 43
2.3.4 获得准确的测试结果 44
2.3.5 运行基准测试并分析结果 46
2.3.6 绘图的重要性 47
2.4 基准测试工具 49
2.4.1 集成式测试工具 49
2.4.2 单组件式测试工具 50
2.5 基准测试案例 52
2.5.1 http_load 53
2.5.2 MySQL 基准测试套件 54
2.5.3 sysbench 55
2.5.4 数据库测试套件中的 dbt2 TPC-C 测试 60
2.5.5 Percona 的 TPCC-MySQL 测试工具 63
2.6 总结 65

第 3 章 服务器性能剖析 67

3.1 性能优化简介 67
3.1.1 通过性能剖析进行优化 69
3.1.2 理解性能剖析 71

3.2 对应用程序进行性能剖析 72
3.2.1 测量 PHP 应用程序 74
3.3 剖析 MySQL 查询 77
3.3.1 剖析服务器负载 77
3.3.2 剖析单条查询 81
3.3.3 使用性能剖析 87
3.4 诊断间歇性问题 88
3.4.1 单条查询问题还是服务器问题 89
3.4.2 捕获诊断数据 93
3.4.3 一个诊断案例 98
3.5 其他剖析工具 106
3.5.1 使用 USER_STATISTICS 表 106
3.5.2 使用 strace 107
3.6 总结 108

第 4 章 Schema 与数据类型优化 111

4.1 选择优化的数据类型 111
4.1.1 整数类型 113
4.1.2 实数类型 113
4.1.3 字符串类型 114
4.1.4 日期和时间类型 121
4.1.5 位数据类型 123
4.1.6 选择标识符（identifier）........125
4.1.7 特殊类型数据 127
4.2 MySQL schema 设计中的陷阱 127
4.3 范式和反范式 129
4.3.1 范式的优点和缺点 130
4.3.2 反范式的优点和缺点 130
4.3.3 混用范式化和反范式化 131
4.4 缓存表和汇总表 132
4.4.1 物化视图 134
4.4.2 计数器表 135
4.5 加快 ALTER TABLE 操作的速度 136

4.5.1 只修改 .frm 文件 137
4.5.2 快速创建 MyISAM 索引 139
4.6 总结 140

第 5 章 创建高性能的索引 141

5.1 索引基础 141
5.1.1 索引的类型 142
5.2 索引的优点 152
5.3 高性能的索引策略 153
5.3.1 独立的列 153
5.3.2 前缀索引和索引选择性 153
5.3.3 多列索引 157
5.3.4 选择合适的索引列顺序 159
5.3.5 聚簇索引 162
5.3.6 覆盖索引 171
5.3.7 使用索引扫描来做排序 175
5.3.8 压缩（前缀压缩）索引 177
5.3.9 冗余和重复索引 178
5.3.10 未使用的索引 181
5.3.11 索引和锁 181
5.4 索引案例学习 183
5.4.1 支持多种过滤条件 183
5.4.2 避免多个范围条件 185
5.4.3 优化排序 186
5.5 维护索引和表 187
5.5.1 找到并修复损坏的表 187
5.5.2 更新索引统计信息 188
5.5.3 减少索引和数据的碎片 190
5.6 总结 192

第 6 章 查询性能优化 195

6.1 为什么查询速度会慢 195

6.2 慢查询基础：优化数据访问 196
6.2.1 是否向数据库请求了不需要的数据 196
6.2.2 MySQL 是否在扫描额外的记录 198
6.3 重构查询的方式 201
6.3.1 一个复杂查询还是多个简单查询 201
6.3.2 切分查询 202
6.3.3 分解关联查询 203
6.4 查询执行的基础 204
6.4.1 MySQL 客户端 / 服务器通信协议 205
6.4.2 查询缓存 208
6.4.3 查询优化处理 208
6.4.4 查询执行引擎 222
6.4.5 返回结果给客户端 223
6.5 MySQL 查询优化器的局限性 223
6.5.1 关联子查询 223
6.5.2 UNION 的限制 228
6.5.3 索引合并优化 228
6.5.4 等值传递 229
6.5.5 并行执行 229
6.5.6 哈希关联 229
6.5.7 松散索引扫描 229
6.5.8 最大值和最小值优化 231
6.5.9 在同一个表上查询和更新 232
6.6 查询优化器的提示（hint） 232
6.7 优化特定类型的查询 236
6.7.1 优化 COUNT() 查询 236
6.7.2 优化关联查询 239
6.7.3 优化子查询 239
6.7.4 优化 GROUP BY 和 DISTINCT 239
6.7.5 优化 LIMIT 分页 241
6.7.6 优化 SQL_CALC_FOUND_ROWS 243
6.7.7 优化 UNION 查询 243
6.7.8 静态查询分析 244

6.7.9 使用用户自定义变量......244
6.8 案例学习......251
6.8.1 使用 MySQL 构建一个队列表......251
6.8.2 计算两点之间的距离......254
6.8.3 使用用户自定义函数......257
6.9 总结......258

第 7 章 MySQL 高级特性......259
7.1 分区表......259
7.1.1 分区表的原理......260
7.1.2 分区表的类型......261
7.1.3 如何使用分区表......262
7.1.4 什么情况下会出问题......263
7.1.5 查询优化......266
7.1.6 合并表......267
7.2 视图......270
7.2.1 可更新视图......272
7.2.2 视图对性能的影响......273
7.2.3 视图的限制......274
7.3 外键约束......275
7.4 在 MySQL 内部存储代码......276
7.4.1 存储过程和函数......278
7.4.2 触发器......279
7.4.3 事件......281
7.4.4 在存储程序中保留注释......283
7.5 游标......283
7.6 绑定变量......284
7.6.1 绑定变量的优化......286
7.6.2 SQL 接口的绑定变量......286
7.6.3 绑定变量的限制......288
7.7 用户自定义函数......289
7.8 插件......290
7.9 字符集和校对......291

7.9.1 MySQL 如何使用字符集......292
7.9.2 选择字符集和校对规则......295
7.9.3 字符集和校对规则如何影响查询......296
7.10 全文索引......299
7.10.1 自然语言的全文索引......300
7.10.2 布尔全文索引......302
7.10.3 MySQL 5.1 中全文索引的变化......303
7.10.4 全文索引的限制和替代方案......304
7.10.5 全文索引的配置和优化......306
7.11 分布式（XA）事务......307
7.11.1 内部 XA 事务......307
7.11.2 外部 XA 事务......308
7.12 查询缓存......309
7.12.1 MySQL 如何判断缓存命中......309
7.12.2 查询缓存如何使用内存......311
7.12.3 什么情况下查询缓存能发挥作用......313
7.12.4 如何配置和维护查询缓存......316
7.12.5 InnoDB 和查询缓存......319
7.12.6 通用查询缓存优化......320
7.12.7 查询缓存的替代方案......321
7.13 总结......321

第 8 章 优化服务器设置......325

8.1 MySQL 配置的工作原理......326
8.1.1 语法、作用域和动态性......327
8.1.2 设置变量的副作用......328
8.1.3 入门......331
8.1.4 通过基准测试迭代优化......332
8.2 什么不该做......333
8.3 创建 MySQL 配置文件......335
8.3.1 检查 MySQL 服务器状态变量......339
8.4 配置内存使用......340
8.4.1 MySQL 可以使用多少内存......340

8.4.2 每个连接需要的内存......341
8.4.3 为操作系统保留内存......341
8.4.4 为缓存分配内存......342
8.4.5 InnoDB 缓冲池（Buffer Pool）......342
8.4.6 MyISAM 键缓存（Key Caches）......344
8.4.7 线程缓存......346
8.4.8 表缓存（Table Cache）......347
8.4.9 InnoDB 数据字典（Data Dictionary）......348
8.5 配置 MySQL 的 I/O 行为......349
8.5.1 InnoDB I/O 配置......349
8.5.2 MyISAM 的 I/O 配置......361
8.6 配置 MySQL 并发......363
8.6.1 InnoDB 并发配置......364
8.6.2 MyISAM 并发配置......365
8.7 基于工作负载的配置......366
8.7.1 优化 BLOB 和 TEXT 的场景......367
8.7.2 优化排序（Filesorts）......368
8.8 完成基本配置......369
8.9 安全和稳定的设置......371
8.10 高级 InnoDB 设置......374
8.11 总结......376

第 9 章 操作系统和硬件优化......377

9.1 什么限制了 MySQL 的性能......377
9.2 如何为 MySQL 选择 CPU......378
9.2.1 哪个更好：更快的 CPU 还是更多的 CPU......378
9.2.2 CPU 架构......380
9.2.3 扩展到多个 CPU 和核心......381
9.3 平衡内存和磁盘资源......382
9.3.1 随机 I/O 和顺序 I/O......383
9.3.2 缓存，读和写......384
9.3.3 工作集是什么......385
9.3.4 找到有效的内存 / 磁盘比例......386

9.3.5 选择硬盘 387
9.4 固态存储 389
9.4.1 闪存概述 390
9.4.2 闪存技术 391
9.4.3 闪存的基准测试 392
9.4.4 固态硬盘驱动器（SSD） 393
9.4.5 PCIe 存储设备 395
9.4.6 其他类型的固态存储 396
9.4.7 什么时候应该使用闪存 396
9.4.8 使用 Flashcache 397
9.4.9 优化固态存储上的 MySQL 399
9.5 为备库选择硬件 402
9.6 RAID 性能优化 403
9.6.1 RAID 的故障转移、恢复和镜像 405
9.6.2 平衡硬件 RAID 和软件 RAID 406
9.6.3 RAID 配置和缓存 407
9.7 SAN 和 NAS 410
9.7.1 SAN 基准测试 411
9.7.2 使用基于 NFS 或 SMB 的 SAN 412
9.7.3 MySQL 在 SAN 上的性能 412
9.7.4 应该用 SAN 吗 413
9.8 使用多磁盘卷 414
9.9 网络配置 416
9.10 选择操作系统 418
9.11 选择文件系统 419
9.12 选择磁盘队列调度策略 421
9.13 线程 422
9.14 内存交换区 422
9.15 操作系统状态 424
9.15.1 如何阅读 vmstat 的输出 425
9.15.2 如何阅读 iostat 的输出 426
9.15.3 其他有用的工具 428
9.15.4 CPU 密集型的机器 428

9.15.5 I/O 密集型的机器 429
9.15.6 发生内存交换的机器 430
9.15.7 空闲的机器 430
9.16 总结 431

第 10 章 复制 433

10.1 复制概述 433
10.1.1 复制解决的问题 434
10.1.2 复制如何工作 435
10.2 配置复制 436
10.2.1 创建复制账号 437
10.2.2 配置主库和备库 437
10.2.3 启动复制 439
10.2.4 从另一个服务器开始复制 441
10.2.5 推荐的复制配置 443
10.3 复制的原理 445
10.3.1 基于语句的复制 445
10.3.2 基于行的复制 446
10.3.3 基于行或基于语句：哪种更优 446
10.3.4 复制文件 448
10.3.5 发送复制事件到其他备库 449
10.3.6 复制过滤器 450
10.4 复制拓扑 452
10.4.1 一主库多备库 452
10.4.2 主动 - 主动模式下的主 - 主复制 453
10.4.3 主动 - 被动模式下的主 - 主复制 455
10.4.4 拥有备库的主 - 主结构 456
10.4.5 环形复制 457
10.4.6 主库、分发主库以及备库 458
10.4.7 树或金字塔形 460
10.4.8 定制的复制方案 460
10.5 复制和容量规划 465
10.5.1 为什么复制无法扩展写操作 466

10.5.2 备库什么时候开始延迟 466
10.5.3 规划冗余容量 467
10.6 复制管理和维护 468
10.6.1 监控复制 468
10.6.2 测量备库延迟 469
10.6.3 确定主备是否一致 469
10.6.4 从主库重新同步备库 470
10.6.5 改变主库 471
10.6.6 在一个主 - 主配置中交换角色 476
10.7 复制的问题和解决方案 477
10.7.1 数据损坏或丢失的错误 477
10.7.2 使用非事务型表 480
10.7.3 混合事务型和非事务型表 480
10.7.4 不确定语句 481
10.7.5 主库和备库使用不同的存储引擎 481
10.7.6 备库发生数据改变 481
10.7.7 不唯一的服务器 ID 482
10.7.8 未定义的服务器 ID 482
10.7.9 对未复制数据的依赖性 482
10.7.10 丢失的临时表 483
10.7.11 不复制所有的更新 484
10.7.12 InnoDB 加锁读引起的锁争用 484
10.7.13 在主 - 主复制结构中写入两台主库 486
10.7.14 过大的复制延迟 488
10.7.15 来自主库的过大的包 491
10.7.16 受限制的复制带宽 491
10.7.17 磁盘空间不足 492
10.7.18 复制的局限性 492
10.8 复制有多快 492
10.9 MySQL 复制的高级特性 494
10.10 其他复制技术 496
10.11 总结 498

第 11 章 可扩展的 MySQL 501

11.1 什么是可扩展性 501
11.1.1 正式的可扩展性定义 503
11.2 扩展 MySQL 507
11.2.1 规划可扩展性 507
11.2.2 为扩展赢得时间 508
11.2.3 向上扩展 509
11.2.4 向外扩展 510
11.2.5 通过多实例扩展 525
11.2.6 通过集群扩展 526
11.2.7 向内扩展 530
11.3 负载均衡 532
11.3.1 直接连接 534
11.3.2 引入中间件 537
11.3.3 一主多备间的负载均衡 540
11.4 总结 541

第 12 章 高可用性 543

12.1 什么是高可用性 543
12.2 导致宕机的原因 544
12.3 如何实现高可用性 545
12.3.1 提升平均失效时间（MTBF） 545
12.3.2 降低平均恢复时间（MTTR） 547
12.4 避免单点失效 548
12.4.1 共享存储或磁盘复制 549
12.4.2 MySQL 同步复制 551
12.4.3 基于复制的冗余 555
12.5 故障转移和故障恢复 556
12.5.1 提升备库或切换角色 558
12.5.2 虚拟 IP 地址或 IP 接管 558
12.5.3 中间件解决方案 559
12.5.4 在应用中处理故障转移 560
12.6 总结 560

第 13 章 云端的 MySQL......563

13.1 云的优点、缺点和相关误解......564
13.2 MySQL 在云端的经济价值......566
13.3 云中的 MySQL 的可扩展性和高可用性......567
13.4 四种基础资源......568
13.5 MySQL 在云主机上的性能......569
13.5.1 在云端的 MySQL 基准测试......571
13.6 MySQL 数据库即服务（DBaaS）......573
13.6.1 Amazon RDS......573
13.6.2 其他 DBaaS 解决方案......574
13.7 总结......575

第 14 章 应用层优化......577

14.1 常见问题......577
14.2 Web 服务器问题......579
14.2.1 寻找最优并发度......581
14.3 缓存......582
14.3.1 应用层以下的缓存......583
14.3.2 应用层缓存......584
14.3.3 缓存控制策略......586
14.3.4 缓存对象分层......587
14.3.5 预生成内容......588
14.3.6 作为基础组件的缓存......589
14.3.7 使用 HandlerSocket 和 memcached......589
14.4 拓展 MySQL......590
14.5 MySQL 的替代品......590
14.6 总结......591

第 15 章 备份与恢复......593

15.1 为什么要备份......594
15.2 定义恢复需求......595
15.3 设计 MySQL 备份方案......596

15.3.1 在线备份还是离线备份........597
15.3.2 逻辑备份还是物理备份........598
15.3.3 备份什么........601
15.3.4 存储引擎和一致性........603
15.4 管理和备份二进制日志........605
15.4.1 二进制日志格式........606
15.4.2 安全地清除老的二进制日志........607
15.5 备份数据........607
15.5.1 生成逻辑备份........607
15.5.2 文件系统快照........610
15.6 从备份中恢复........617
15.6.1 恢复物理备份........618
15.6.2 还原逻辑备份........619
15.6.3 基于时间点的恢复........622
15.6.4 更高级的恢复技术........624
15.6.5 InnoDB 崩溃恢复........625
15.7 备份和恢复工具........628
15.7.1 MySQL Enterprise Backup........628
15.7.2 Percona XtraBackup........628
15.7.3 mylvmbackup........629
15.7.4 Zmanda Recovery Manager........629
15.7.5 mydumper........629
15.7.6 mysqldump........629
15.8 备份脚本化........631
15.9 总结........633

第 16 章 MySQL 用户工具........635

16.1 接口工具........635
16.2 命令行工具集........636
16.3 SQL 实用集........637
16.4 监测工具........637
16.4.1 开源的监控工具........638
16.4.2 商业监控系统........640

16.4.3 Innotop 的命令行监控 642
16.5 总结 646

附录 A MySQL 分支与变种 649

附录 B MySQL 服务器状态 655

附录 C 大文件传输 683

附录 D EXPLAIN 687

附录 E 锁的调试 703

附录 F 在 MySQL 上使用 Sphinx 713

索引 739

推荐序

很多年前我就是这本书的“粉丝”了，这是一本伟大的书，第三版尤其如此。这些世界级的专家不仅仅分享他们的专业知识，也花了很多时间来更新和添加新的章节，且都是高品质的内容。本书有大量关于如何获得 MySQL 高性能的细节信息，并且关注的是提升性能的过程，而不仅仅是描述事实结果和琐碎的细枝末节。这本书将告诉读者如何将事情做得更好，不管 MySQL 在不同版本中的行为有多么大的改变。

毫无疑问，本书的作者是唯一有资格来写这么一本书的人，他们经验丰富，有合理的方法，关注效率，并且精益求精。说到经验丰富，本书的作者已经在 MySQL 性能领域工作多年，从 MySQL 还没有什么可扩展性和可测量性的时代，直到现在这些方面已经有了长足的进步。而说到合理的方法，他们简直把这件事情当成了科学，首先定义需要解决的问题，然后通过合理的猜测和精确的测量来解决问题。

我对作者在效率方面的关注尤其印象深刻。作为顾问，他们时间宝贵。客户是按照他们的时间付费的，所以都希望能更快地解决问题。所以本书作者定义了一整套的流程，开发了很多的工具，让事情变得正确和高效。在本书中，作者详细描述了这些流程，并且发布了工具的源代码。

最后，本书作者在工作上一直精益求精。比如从吞吐量到响应时间的关注，致力于了解 MySQL 在新硬件上的性能表现，追求新的技能如排队理论对性能的影响，等等。

我相信本书预示了 MySQL 的光明前景。MySQL 已经支持高要求的工作负载，本书作者也在努力提升 MySQL 社区内对性能的认识。同时，他们还直接为性能提升做出了贡献，包括 XtraDB 和 XtraBackup。一直以来我从他们身上学到了不少东西，也希望读者多花点时间读读本书，一定会同样有所收益。

——Mark Callaghan，Facebook 软件工程师

前言

我们写这本书不仅仅是为了满足 MySQL 应用开发者的需求，也是为了满足 MySQL 数据库管理员的需要。我们假定读者已经有了一定的 MySQL 基础。我们还假定读者对于系统管理、网络和类 Unix 的操作系统都有一些了解。

本书的第二版为读者提供了大量的信息，但没有一本书是可以涵盖一个主题的所有方面的。在第二版和第三版之间的这段时间里，我们记录了数以千计有趣的问题，其中有些是我们解决的，也有一些是我们观察到其他人解决的。当我们在规划第三版的时候发现，如果要把这些主题完全覆盖，可能三千页到五千页的篇幅都还不够，这样本书的完成就遥遥无期了。在反思这个问题后，我们意识到第二版强调的广泛的覆盖度事实上有其自身的限制，从某种意义上来说也没有引导读者如何按照 MySQL 的方式来思考问题。

所以第三版和第二版的关注点有很大的不同。我们虽然还是会包含很多的信息，并且会强调同样的诸如可靠性和正确性的目标，但我们也会在本书中尝试更深入的讨论：我们会指出 MySQL 为什么会这样做，而不是 MySQL 做了什么。我们会使用更多的演示和案例学习来将上述原则落地。通过这样的方式，我们希望能够尝试回到下面这样的问题："给出 MySQL 的内部结构和操作，对于实际应用能带来什么帮助？为什么能有这样的帮助？如何让 MySQL 适合（或者不适合）特定的需求？"

最后，我们希望关于 MySQL 内部原理的知识能够帮助大家解决本书没有覆盖到的一些情况。我们更希望读者能培养发现新问题的洞察力，能学习和实践合理的方式来设计、维护和诊断基于 MySQL 的系统。

本书是如何组织的

本书涵盖了许多复杂的主题。在这里，我们将解释一下是如何将这些主题有序地组织在一起的，以便于阅读和学习。

概述

第 1 章是非常基础的一章，在更深入地学习之前建议先熟悉一下这部分内容。在有效地使用 MySQL 之前应当理解它是如何组织的。本章解释了 MySQL 的架构及其存储引擎的关键设计。如果读者还不太熟悉关系数据库和事务的基础知识，本章也可以带来一点帮助。如果之前已经对其他关系数据库如 Oracle 比较熟悉，本章也可以帮助读者了解 MySQL 的入门知识。本章还包括了一点 MySQL 的历史背景：MySQL 随着时间的演进、最近的公司所有权更替，以及我们认为比较重要的内容。

打造坚实的基础

本书前几章的内容在今后使用 MySQL 的过程中可能会被不断地引用到，它们是非常基础的内容。

第 2 章讨论了基准测试的基础，例如服务器可以处理的工作负载的类型、处理特定任务的速度等。基准测试是一项至关重要的技能，可用于评估服务器在不同负载下的表现，但也要明白在什么情况下基准测试不能发挥作用。

第 3 章介绍了我们常用于故障诊断和服务器性能问题分析的一种面向响应时间的方法。该方法已经被证明可以解决我们曾碰到过的一些极为棘手的问题。当然也可以选择修改我们所使用的方法（实际上我们的方法也是从 Cary Millsap 的方法修改而来的），但无论如何，至少不能没有方法胡乱猜测。

从第 4 章到第 6 章，连续介绍了三个关于良好的数据库逻辑设计和物理设计基础的话题。第 4 章涵盖了不同数据类型的细节差别以及表设计的原则。第 5 章则展开讨论了索引，这是数据库的物理设计。对于索引的深入理解和利用是高效使用 MySQL 的基础，相信这一章会经常需要回头翻看。而第 6 章则包含了分析 MySQL 的查询是如何执行的，以及如何利用查询优化器的话题。该章也包含了大量常见类型查询的例子，演示了 MySQL 是如何做好工作的，以及如何改写查询以利用 MySQL 的特性。

到此为止，已经覆盖了关于数据库的基础内容：表、索引、数据和查询。第 7 章则在 MySQL 基础知识之外介绍了 MySQL 的高级特性是如何工作的。这章的内容包括分区、存储引擎、触发器，以及字符集。MySQL 中这些特性的实现可能不同于其他数据库，可能之前读者并不清楚这些不同，因此理解它们对于性能可能会带来新的收益。

配置应用程序

接下来的两章讲述的是如何让 MySQL、应用程序及硬件一起很好地工作。第 8 章介绍了如何配置 MySQL，以便更好地利用硬件，达到更好的可靠性和鲁棒性。第 9 章解释

了如何让操作系统和硬件工作得更好。另外也深入讨论了固态硬盘，为高可扩展性应用发挥更好的性能提供了硬件配置的建议。

上面两章都一定程度地涉及了 MySQL 的内部知识。这将会是一个反复出现的主题，附录中也会有相关内容可以学习到 MySQL 的内部是如何实现的，理解了这些知识将帮助读者更好地理解某些现象背后的原理。

作为基础设施组件的 MySQL

MySQL 不是存在于真空中的，而是应用整体的一个环节，因此需要考虑整个应用架构的鲁棒性。下面的章节将告诉我们该如何做到这一点。

第 10 章讨论了 MySQL 的杀手级特性：能够设置多个服务器从一台主服务器同步数据。不幸的是，复制可能也是 MySQL 给很多用户带来困扰的一个特性。但实际上不应该发生这样的情况，本章将告诉你如何让复制运行得更好。

第 11 章讨论了什么是可扩展性（这和性能不是一回事），应用和系统为什么会无法扩展，该怎么改善扩展性。如果能够正确地处理，MySQL 的可扩展性是足以应付任何需求的。第 12 章讲述的是和可扩展性相关但又完全不同的主题：如何保障 MySQL 稳定而正确地持续运行。第 13 章将告诉你当 MySQL 在云计算环境中运行时会有什么不同的事情发生。

第 14 章解释了什么是全方位的优化（full-stack optimization），就是从前端到后端的整体优化，从用户体验开始直到数据库。

即使是世界上设计最好、最具可扩展性的架构，如果停电会导致彻底崩溃，无法抵御恶意攻击，解决不了应用的 bug 和程序员的错误，以及其他一些灾难场景，那就不是什么好的架构。第 15 章讨论了 MySQL 数据库各种备份与恢复的场景。这些策略可以帮助读者减少在各种不可抗的硬件失效时的宕机时间，保证在各种灾难下的数据最终可恢复。

其他有用的主题

在本书的最后一章以及附录中，我们探讨了一些无法明确地放到前面章节的内容，以及一些被前面多个章节引用而需要特别注意的主题。

第 16 章探索了一些可以帮助用户更有效地管理和监控 MySQL 服务器的工具，有些是开源的，也有些是商业的。

附录 A 介绍了近年来成长迅速的三个主要的非 MySQL 官方版本，其中一个是我们公司在维护的产品。知道还有其他什么是可用的选择是有价值的；很多 MySQL 难以解决的棘手问题在其他的变种版本中说不定就不是问题了。这三个版本中的两个（Percona

Server 和 MariaDB）是 MySQL 的完全可替换版本，所以尝试使用的成本相对来说是很低的。当然，在这里我们也需要补充一点，Oracle 提供的 MySQL 官方版本对于大多数用户来说都能服务得很好。

附录 B 演示了如何检查 MySQL 服务器。知道如何从服务器获取状态信息是非常重要的；而了解这些状态代表的意义则更加重要。这里将覆盖 `SHOW INNODB STATUS` 的输出结果，因此这里包含了 InnoDB 事务存储引擎的深入信息。在这个附录中讨论了很多 InnoDB 的内部信息。

附录 C 演示了如何高效地将大文件从一个地方复制到另外一个地方。如果要管理大量的数据，这种操作是经常都会碰到的。附录 D 演示了如何真正地使用并理解 `EXPLAIN` 命令。附录 E 演示了如何破除不同查询所请求的锁互相干扰的问题。最后，附录 F 介绍了 Sphinx，一个基于 MySQL 的高性能的全文索引系统。

软件版本与可用性

MySQL 是一个移动靶。从 Jeremy 写作本书第一版到现在，MySQL 已经发布了好几个版本。当本书第一版的初稿交给出版社的时候，MySQL 4.1 和 5.0 还只是 alpha 版本，而如今 MySQL 5.1 和 5.5 已经是很多在线应用的主力版本。在我们写完这第三版的时候，MySQL 5.6 也即将发布。

本书的内容并不依赖某一个具体的版本。相反，我们会利用自己在实际环境中获得的更广泛的知识。本书的核心内容主要关注 MySQL 5.1 和 5.5 版本，因为我们认为这是“当前”的版本。本书的大多数例子都假设运行在 MySQL 5.1 的某个成熟版本上，比如 MySQL 5.1.50 或者更高的版本。对于在旧版本中可能不存在，或者只在即将到来的 5.6 版本中出现的特性或者功能，我们也会特别标注出来。然而，关于某个 MySQL 版本的特性的权威指南还是要看官方文档。在阅读本书时，建议随时访问在线官方文档的相关内容（*http://dev.mysql.com/doc/*）。

MySQL 的另外一个伟大特点是能够运行在现今流行的所有平台：Mac OS X，Windows, GNU/Linux, Solaris, FreeBSD，以及只要你能举出名字的其他平台。然而，本书主要基于 GNU/Linux[注1] 和其他类 Unix 系统。Windows 的用户可能会碰到一些困难。比如说文件路径就和 Windows 完全不一样。我们也会引用一些 Unix 的命令行工具，我们假设读者能够知道 Windows 上对应的工具是什么[注2]。

注 1：为了避免产生疑惑，如果我们指的是内核的时候用的是 Linux，如果指的是支持应用的整个操作系统环境的时候用的是 GNU/Linux。

注 2：可以从 *http://unxutils.sourceforge.net* 或者 *http://gnuwin32.sourceforge.net* 获得 Unix 工具的 Windows 兼容版本。

在 Windows 上搞 MySQL 的另外一个难点是 Perl。MySQL 中有很多有用的工具是用 Perl 写的。在本书的一些章节中，也有一些 Perl 脚本，在此基础上可以构建更加复杂的工具。Percona Toolkit 是不可多得的 MySQL 管理工具，也是用 Perl 写的。然而，Windows 平台默认是没有 Perl 环境的。为了使用这些工具，需要从 ActiveState 下载 Perl 的 Windows 版本，以及访问 MySQL 所需要的一些额外的模块（`DBI` 和 `DBD::MySQL`）。

本书使用的约定

下面是本书中使用的一些约定。

斜体（*Italic*）

  新的名字、URL、邮件地址、用户名、主机名、文件名、文件扩展名、路径名、目录，以及 Unix 命令和工具都使用斜体表示。

等宽字体（`Constant width`）

  包括代码元素、配置选项、数据库和表名、变量和值、函数、模块、文件内容、命令输出等，使用的是等宽字体。

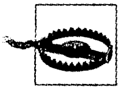
加粗的等宽字体（**`Constant width bold`**）

  命令或者其他需要用户输入的文本，命令输出中需要强调的某些内容，会使用加粗的等宽字体。

斜体的等宽字体（*`Constant width italic`*）

  需要用户替换的文本以斜体的等宽字体表示。

这个图标表示提示、建议，或者一般的记录。



这个图标表示一个警告或者提醒。

使用示例代码

本书的目标是为了帮助读者更好地工作。一般来说，你可以在程序或者文档中使用本书中的代码。只要不是大规模地复制重要的代码，使用的时候不需要联系我们。例如，你编写的程序中如果只是使用了本书部分的代码片段则无须取得授权，而出售或者分发 O'Reilly 书籍示例代码的 CD-ROM 盘片则需要经过授权。引用本书的代码回答问题也无须取得授权，而大量引用本书的示例代码到产品文档中则需要获取授权。

示例代码维护在 *http://www.highperfmysql.com* 站点中，会及时保持更新。但我们无法确保代码会跟随每一个 MySQL 的小版本进行更新和测试。

我们欢迎大家在使用了本书代码后进行反馈，但这不是一个强制要求。反馈时请提供标题、作者、出版公司和 ISBN。例如："*High Performance MySQL, Third Edition*, by Baron Schwartz et al. (O'Reilly). Copyright 2012 Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko, 978-1-449-31428-6"。

如果你使用了本书的代码，但又不在上面描述的一些无须授权的范围之内，不确定是否需要获取授权时，请联系 *permissions@oreilly.com*。

Safari 在线书店

Safari 在线书店（*www.safaribooksonline.com*）是一家提供定制服务的数字图书馆，提供技术和商务领域内顶级作家的高质量内容的书籍和音像制品。很多技术专家、软件开发者、Web 设计师、商务人士和创新专家都将 Safari 在线书店作为他们研究、解决问题、学习和认证练习的首选资料来源。

Safari 在线书店为组织、政府机构和个人提供了一系列的产品组合和定价计划。订阅者可以访问数以千计的图书、培训视频和手稿，这些存在于一个可搜索的数据库中，涵盖的出版公司有 O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology，等等。如需了解更多关于 Safari 在线书店的情况，请访问在线网站。

如何联系我们

若有关于本书的任何评论或者问题，请和出版公司联系。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

本书有一个配套的网页，上面列出了勘误表、示例代码及其他相关信息。下面是此网页的地址：

http://shop.oreilly.com/product/0636920022343.do

如果有关于本书的评论和技术问题，也可以通过邮件进行沟通：

bookquestions@oreilly.com

如果想了解更多关于我们出版公司的书籍、会议、资源中心和 O'Reilly 网络的信息，请访问网站：

http://www.oreilly.com

我们的 Facebook：*http://facebook.com/oreilly*

我们的 Twitter：*http://twitter.com/oreillymedia*

我们的 YouTube：*http://www.youtube.com/oreillymedia*

当然，读者也可以直接和作者取得联系，可以访问作者的公司网站 *http://www.percona.com*。我们将乐于收到大家的反馈。

本书第三版的致谢

感谢以下人员给予的各种帮助：Brian Aker, Johan Andersson, Espen Braekken, Mark Callaghan, James Day, Maciej Dobrzanski, Ewen Fortune, Dave Hildebrandt, Fernando Ipar, Haidong Ji, Giuseppe Maxia, Aurimas Mikalauskas, Istvan Podor, Yves Trudeau, Matt Yonkovit, Alex Yurchenko。感谢 Percona 公司的所有员工，多年来为本书提供了无数的支持。感谢很多著名博主[注3]和技术大会的演讲者，他们为本书的很多思想提供了大量的素材，尤其是 Yoshinori Matsunobu。另外也要感谢本书前面两版的作者：Jeremy D. Zawodny、Derek J. Balling 和 Arjen Lentz。感谢 Andy Oram、Rachel Head，以及 O'Reilly 的整个编辑团队，你们为本书的出版和发行做了卓有成效的工作。非常感谢 Oracle 的才华横溢且专注的 MySQL 团队，以及所有之前的 MySQL 开发者，不管你现在是在 SkySQL 还是在 Monty 团队。

Baron 也要感谢他的妻子 Lynn、他的母亲 Connie，以及他的岳父母 Jane 和 Roger，感谢他们一如既往地支持他的工作，尤其是不断地鼓励他，并且承担了所有的家务和照顾整个家庭的重任。也要感谢 Peter 和 Vadim，你们是如此优秀的老师和同事。Baron 将此版

注 3： 在 *http://planet.mysql.com* 网站上可以找到很多优秀的技术博客。

本献给 Alan Rimm-Kaufman，以纪念他给予的伟大的爱和鼓励，这些都将永志不忘。

本书第二版的致谢

Sphinx 的开发者 Andrew Aksyonoff 编写了附录 F。我们非常感谢他首次对此进行深入的讨论。

在编写本书的时候，我们得到了很多人的无私帮助。在此无法一一列举——我们真的非常感谢 MySQL 社区和 MySQL AB 公司的每一个人。下面是对本书做出了直接贡献的人，如有遗漏，还请见谅。他们是：Tobias Asplund, Igor Babaev, Pascal Borghino, Roland Bouman, Ronald Bradford, Mark Callaghan, Jeremy Cole, Britt Crawford 和他的 HiveDB 项目，Vasil Dimov, Harrison Fisk, Florian Haas, Dmitri Joukovski 和他的 Zmanda 项目（同时感谢 Dmitri 为解释 LVM 快照提供的配图），Alan Kasindorf, Sheeri Kritzer Cabral, Marko Makela, Giuseppe Maxia, Paul McCullagh, B. Keith Murphy, Dhiren Patel, Sergey Petrunia, Alexander Rubin, Paul Tuckfield, Heikki Tuuri, 以及 Michael "Monty" Widenius。

在这里还要特别感谢 O'Reilly 的编辑 Andy Oram 和助理编辑 Isabel Kunkle，以及审稿人 Rachel Wheeler，同时也要感谢 O'Reilly 团队的其他所有成员。

来自 Baron

我要感谢我的妻子 Lynn Rainville 和小狗 Carbon。如果你也曾写过一本书，我相信你就能体会到我是如何地感谢他们。我也非常感谢 Alan Rimm-Kaufman 和我在 Rimm-Kaufman 集团的同事，在写书的过程中，他们给了我支持和鼓励。谢谢 Peter、Vadim 和 Arjen，是你们给了我梦想成真的机会。最后，我要感谢 Jeremy 和 Derek 为我们开了个好头。

来自 Peter

我从事 MySQL 性能和可扩展性方面的演讲、培训和咨询工作已经很多年了，我一直想把它们扩展到更多的受众。因此，当 Andy Oram 加入到本书的编写当中时，我感到非常兴奋。此前我没有写过书，所以我对所需要的时间和精力都毫无把握。一开始我们谈到只对第一版做一些更新，以跟上 MySQL 最新的版本升级，但我们想把更多新素材加入到书中，结果几乎相当于重写了整本书。

这本书是真正的团队合作的结晶。因为我忙于 Percona 公司的事情——这是我和 Vadim 的咨询公司，而且英语并非我的第一语言，所以我们有着不同的角色分工。我负责提供大纲和技术性内容，评审所有的材料，在写作的时候再进行修订和扩展。当 Arjen（MySQL 文档团队的前负责人）加入之后，我们就开始勾画出整个提纲。在 Baron 加入后，一切

才开始真正行动起来，他能够以不可思议的速度编写出高质量的内容。Vadim 则在深入检查 MySQL 源代码和提供基准测试及其他研究来巩固我们的论点时提供了巨大的帮助。

当我们编写本书时，我们发现有越来越多的领域需要刨根问底。本书的大部分主题，如复制、查询优化、InnoDB、架构和设计都足以单独成书。因此，有时候我们不得不在某个点停止深入，把余下的材料用在将来可能出版的新版本中，或者我们的博客、演讲和技术文章中。

本书的评审者给了我们非常大的帮助，无论是来自 MySQL AB 公司内部的人员，还是外部的人员，他们都是 MySQL 领域最优秀的世界级专家。其中包括 MySQL 的创建者 Michael Widenius、InnoDB 的创建者 Heikki Tuuri、MySQL 优化器团队的负责人 Igor Babaev，以及其他人。

我还要感谢我的妻子 Katya Zaytseva，我的孩子 Ivan 和 Nadezhda，他们允许我把家庭时间花在了本书的写作上。我也要感谢 Percona 的员工，当我在公司里"人间蒸发"去写书时，他们承担了日常事务的处理工作。当然，我也要感谢 O'Reilly 和 Andy Oram 让这一切成为可能。

来自 Vadim

我要感谢 Peter，能够在本书中和他合作，我感到十分开心，期望在其他项目中能继续共事。我也要感谢 Baron，他在本书的写作过程中起了很大的作用。还有 Arjen，跟他一起工作非常好玩。我还要感谢我们的编辑 Andy Oram，他抱着十二万分的耐心和我们一起工作。此外，还要感谢 MySQL 团队，是他们创造了这个伟大的软件。我还要感谢我们的客户给予我调优 MySQL 的机会。最后，我要特别感谢我的妻子 Valerie，以及我们的孩子 Myroslav 和 Timur，他们一直支持我，帮助我一步步前进。

来自 Arjen

我要感谢 Andy 的睿智、指导和耐心，感谢 Baron 中途加入到我们当中来、感谢 Peter 和 Vadim 坚实的背景信息和基准测试。也要感谢 Jeremy 和 Derek 在第一版中打下的基础。在我的书上，Derak 题写着："要诚实——这就是我的所有要求"。

我也要感谢我在 MySQL AB 公司时的所有同事，在那里我获得了关于本书主题的大多数知识。在此，我还要特别提到 Monty，我一直认为他是令人自豪的 MySQL 之父，尽管他的公司如今已经成为 Sun 公司的一部分。我要感谢全球 MySQL 社区里的每一个人。

最后同样重要的是，我要感谢我的女儿 Phoebe，在她尚年少的生活舞台上，不用关心什么是 MySQL，也不用考虑 Wiggles 指的是什么东西。从某些方面来讲，无知就是福。它

能给予我们一个全新的视角来看清生命中真正重要的是什么。对于读者，祝愿你们的书架上又增添了一本有用的书，还有，不要忘记你的生活。

本书第一版的致谢

要完成这样一本书的写作，离不开许许多多人的帮助。没有他们的无私援助，你手上的这本书就可能仍然是我们显示器屏幕四周的那一堆贴纸。这是本书的一部分，在这里，我们可以感谢每一个曾经帮助我们脱离困境的人，而无须担心突然奏响的背景音乐催促我们闭上嘴巴赶紧走掉——如同你在电视里看到的颁奖晚会那样。

如果没有编辑 Andy Oram 坚决的督促、请求、央求和支持，我们就无法完成本书。如果要找对于本书最负责的一个人，那就是 Andy。我们真的非常感谢每周一次的唠唠叨叨的会议。

然而，Andy 不是一个人在战斗。在 O'Reilly，还有一批人都参与了将那些小贴纸变成你正在看的这本书的工作。所以我们也要感谢那些在生产、插画和销售环节的人们，感谢你们把这本书变成实体。当然，也要感谢 Tim O'Reilly，是他持久不变地承诺为广大开源软件出版一批业内最好的图书。

最后，我们要把感谢给予那些同意审阅本书不同版本草稿，并告诉我们哪里有错误的人们：我们的评审者。他们把 2003 年假期的一部分时间用在了审阅这些格式粗糙，充满了打字符号、误导性的语句和彻底的数学错误的文本上。我们要感谢（排名不分先后）：Brian "Krow" Aker, Mark "JDBC" Matthews, Jeremy "the other Jeremy" Cole, Mike "VBMySQL.com (*http://vbmysql.com*)" Hillyer, Raymond "Rainman" De Roo, Jeffrey "Regex Master" Friedl, Jason DeHaan, Dan Nelson, Steve "Unix Wiz" Friedl, Kasia "Unix Girl" Trapszo。

来自 Jeremy

我要在此感谢 Andy，是他同意接纳这个项目，并持续不断地鞭策我们加入新的章节内容。Derek 的帮助也非常关键，本书最后的 20% ~ 30% 内容由他一手完成，这使得我们没有错过下一个目标日期。感谢他同意中途加入进来，代替我只能偶尔爆发一下的零星生产力，完成了关于 XML 的烦琐工作、第 10 章、附录 F，以及我丢给他的那些活儿。

我也要感谢我的父母，在多年以前他们就给我买了 Commodore 64 电脑，他们不仅在前 10 年里容忍了我就像要以身相许般的对电子和计算机技术的痴迷，并在之后还成为我不懈学习和探索的支持者。

接下来，我要感谢在过去几年里在 Yahoo！布道推广 MySQL 时遇到的那一群人。跟他们共事，我感到非常愉快。在本书的筹备阶段，Jeffrey Friedl 和 Ray Goldberger 给了我鼓励和反馈意见。在他们之后还有 Steve Morris、James Harvey 和 Sergey Kolychev 容忍了我在 Yahoo! Finance MySQL 服务器上做着看似固定不变的实验，即使这打扰了他们的重要工作。我也要感谢 Yahoo! 的其他成员，是他们帮我发现了 MySQL 上的那些有趣的问题和解决方法。还有，最重要的是要感谢他们对我有足够的信任和信念，让我把 MySQL 用在 Yahoo! 重要和可见的部分业务上。

Adam Goodman，一位出版家和 *Linux Magazine* 的拥有者，他帮助我轻装上阵为技术受众撰写文章，并在 2001 年下半年第一次出版了我的 MySQL 相关的长篇文章。自那以后，他教授给我更多他所能认识到的关于编辑和出版的技能，还鼓励我通过在杂志上开设月度专栏在这条路上继续走下去。谢谢你，Adam。

我要感谢 Monty 和 David，感谢你们与这个世界分享了 MySQL。说到 MySQL AB，也要感谢在那里的其他“牛”人，是他们鼓励我写成本书：Kerry, Larry, Joe, Marten, Brian, Paul, Jeremy, Mark, Harrison, Matt，以及团队中的其他人。他们真的非常棒。

最后，我要感谢我博客的读者，是他们鼓励我撰写基于日常工作的非正式的 MySQL 及其他技术文章。最后同样重要的是，感谢 Goon Squad。

来自 Derek

就像 Jeremy 一样，有太多同样的原因，我也要感谢我的家庭。我要感谢我的父母，是他们不停地鼓励我去写一本书，哪怕他们头脑中都没有任何和它相关的东西。我的祖父母给我上了两堂很有价值的课：美元的含义，以及我跟电脑相爱有多深，他们还借钱给我去购买了我平生第一台电脑：Commodore VIC-20。

我万分感谢 Jeremy 邀请我加入他那旋风般的写作“过山车”中来。这是一个很棒的体验，我希望将来还能跟他一起工作。

我要特别感谢 Raymond De Roo, Brian Wohlgemuth, David Calafrancesco, Tera Doty, Jay Rubin, Bill Catlan, Anthony Howe, Mark O’Neal, George Montgomery, George Barber，以及其他无数耐心听我抱怨的人，我从他们那里了解到我所讲述的内容是否能让门外汉也能理解，或者仅仅得到一个我所希望的笑脸。没有他们，这本书可能也能写出来，但我几乎可以肯定我会在这个过程中疯掉。

第 1 章

MySQL架构与历史

和其他数据库系统相比，MySQL 有点与众不同，它的架构可以在多种不同场景中应用并发挥好的作用，但同时也会带来一点选择上的困难。MySQL 并不完美，却足够灵活，能够适应高要求的环境，例如 Web 类应用。同时，MySQL 既可以嵌入到应用程序中，也可以支持数据仓库、内容索引和部署软件、高可用的冗余系统、在线事务处理系统（OLTP）等各种应用类型。

为了充分发挥 MySQL 的性能并顺利地使用，就必须理解其设计。MySQL 的灵活性体现在很多方面。例如，你可以通过配置使它在不同的硬件上都运行得很好，也可以支持多种不同的数据类型。但是，MySQL 最重要、最与众不同的特性是它的存储引擎架构，这种架构的设计将查询处理（Query Processing）及其他系统任务（Server Task）和数据的存储 / 提取相分离。这种处理和存储分离的设计可以在使用时根据性能、特性，以及其他需求来选择数据存储的方式。

本章概要地描述了 MySQL 的服务器架构、各种存储引擎之间的主要区别，以及这些区别的重要性。另外也会回顾一下 MySQL 的历史背景和基准测试，并试图通过简化细节和演示案例来讨论 MySQL 的原理。这些讨论无论是对数据库一无所知的新手，还是熟知其他数据库的专家，都不无裨益。

1.1 MySQL 逻辑架构

如果能在头脑中构建出一幅 MySQL 各组件之间如何协同工作的架构图，就会有助于深入理解 MySQL 服务器。图 1-1 展示了 MySQL 的逻辑架构图。

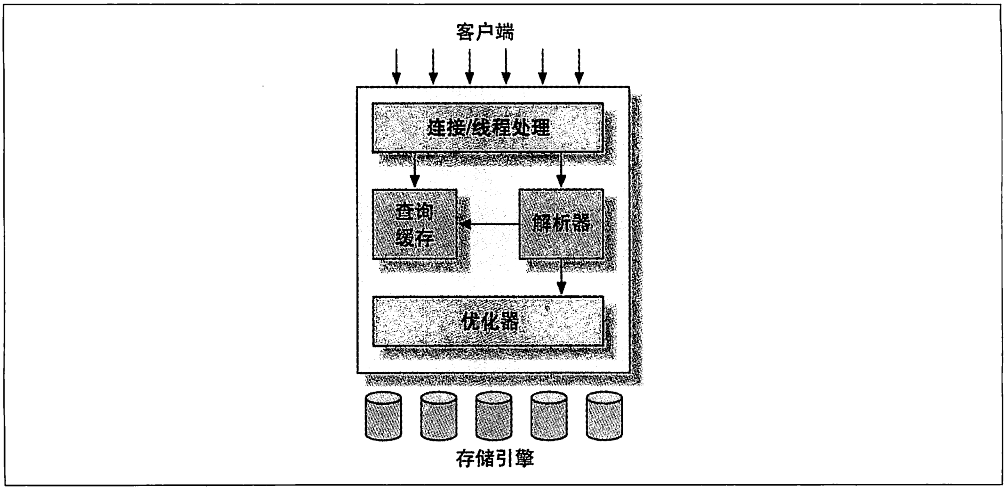


图1-1：MySQL服务器逻辑架构图

最上层的服务并不是 MySQL 所独有的，大多数基于网络的客户端 / 服务器的工具或者服务都有类似的架构。比如连接处理、授权认证、安全等等。

第二层架构是 MySQL 比较有意思的部分。大多数 MySQL 的核心服务功能都在这一层，
2 包括查询解析、分析、优化、缓存以及所有的内置函数（例如，日期、时间、数学和加密函数），所有跨存储引擎的功能都在这一层实现：存储过程、触发器、视图等。

第三层包含了存储引擎。存储引擎负责 MySQL 中数据的存储和提取。和 GNU/Linux 下的各种文件系统一样，每个存储引擎都有它的优势和劣势。服务器通过 API 与存储引擎进行通信。这些接口屏蔽了不同存储引擎之间的差异，使得这些差异对上层的查询过程透明。存储引擎 API 包含几十个底层函数，用于执行诸如“开始一个事务”或者“根据主键提取一行记录”等操作。但存储引擎不会去解析 SQL[注1]，不同存储引擎之间也不会相互通信，而只是简单地响应上层服务器的请求。

1.1.1 连接管理与安全性

每个客户端连接都会在服务器进程中拥有一个线程，这个连接的查询只会在这个单独的线程中执行，该线程只能轮流在某个 CPU 核心或者 CPU 中运行。服务器会负责缓存线程，因此不需要为每一个新建的连接创建或者销毁线程[注2]。

当客户端（应用）连接到 MySQL 服务器时，服务器需要对其进行认证。认证基于用户名、

注 1： InnoDB 是一个例外，它会解析外键定义，因为 MySQL 服务器本身没有实现该功能。

注 2： MySQL 5.5 或者更新的版本提供了一个 API，支持线程池（Thread-Pooling）插件，可以使用池中少量的线程来服务大量的连接。

原始主机信息和密码。如果使用了安全套接字（SSL）的方式连接，还可以使用 X.509 证书认证。一旦客户端连接成功，服务器会继续验证该客户端是否具有执行某个特定查询的权限（例如，是否允许客户端对 world 数据库的 Country 表执行 SELECT 语句）。 3

1.1.2 优化与执行

MySQL 会解析查询，并创建内部数据结构（解析树），然后对其进行各种优化，包括重写查询、决定表的读取顺序，以及选择合适的索引等。用户可以通过特殊的关键字提示（hint）优化器，影响它的决策过程。也可以请求优化器解释（explain）优化过程的各个因素，使用户可以知道服务器是如何进行优化决策的，并提供一个参考基准，便于用户重构查询和 schema、修改相关配置，使应用尽可能高效运行。第 6 章我们将讨论更多优化器的细节。

优化器并不关心表使用的是什么存储引擎，但存储引擎对于优化查询是有影响的。优化器会请求存储引擎提供容量或某个具体操作的开销信息，以及表数据的统计信息等。例如，某些存储引擎的某种索引，可能对一些特定的查询有优化。关于索引与 schema 的优化，请参见第 4 章和第 5 章。

对于 SELECT 语句，在解析查询之前，服务器会先检查查询缓存（Query Cache），如果能够在其中找到对应的查询，服务器就不必再执行查询解析、优化和执行的整个过程，而是直接返回查询缓存中的结果集。第 7 章详细讨论了相关内容。

1.2 并发控制

无论何时，只要有多个查询需要在同一时刻修改数据，都会产生并发控制的问题。本章的目的是讨论 MySQL 在两个层面的并发控制：服务器层与存储引擎层。并发控制是一个内容庞大的话题，有大量的理论文献对其进行过详细的论述。本章只简要地讨论 MySQL 如何控制并发读写，因此读者需要有相关的知识来理解本章接下来的内容。

以 Unix 系统的 email box 为例，典型的 *mbox* 文件格式是非常简单的。一个 *mbox* 邮箱中的所有邮件都串行在一起，彼此首尾相连。这种格式对于读取和分析邮件信息非常友好，同时投递邮件也很容易，只要在文件末尾附加新的邮件内容即可。

但如果两个进程在同一时刻对同一个邮箱投递邮件，会发生什么情况？显然，邮箱的数据会被破坏，两封邮件的内容会交叉地附加在邮箱文件的末尾。设计良好的邮箱投递系统会通过锁（lock）来防止数据损坏。如果客户试图投递邮件，而邮箱已经被其他客户锁住，那就必须等待，直到锁释放才能进行投递。 4

这种锁的方案在实际应用环境中虽然工作良好，但并不支持并发处理。因为在任意一个时刻，只有一个进程可以修改邮箱的数据，这在大容量的邮箱系统中是个问题。

1.2.1 读写锁

从邮箱中读取数据没有这样的麻烦，即使同一时刻多个用户并发读取也不会有什么问题。因为读取不会修改数据，所以不会出错。但如果某个客户正在读取邮箱，同时另外一个用户试图删除编号为 25 的邮件，会产生什么结果？结论是不确定，读的客户可能会报错退出，也可能读取到不一致的邮箱数据。所以，为安全起见，即使是读取邮箱也需要特别注意。

如果把上述的邮箱当成数据库中的一张表，把邮件当成表中的一行记录，就很容易看出，同样的问题依然存在。从很多方面来说，邮箱就是一张简单的数据库表。修改数据库表中的记录，和删除或者修改邮箱中的邮件信息，十分类似。

解决这类经典问题的方法就是并发控制，其实非常简单。在处理并发读或者写时，可以通过实现一个由两种类型的锁组成的锁系统来解决问题。这两种类型的锁通常被称为共享锁（shared lock）和排他锁（exclusive lock），也叫读锁（read lock）和写锁（write lock）。

这里先不讨论锁的具体实现，描述一下锁的概念如下：读锁是共享的，或者说是相互不阻塞的。多个客户在同一时刻可以同时读取同一个资源，而互不干扰。写锁则是排他的，也就是说一个写锁会阻塞其他的写锁和读锁，这是出于安全策略的考虑，只有这样，才能确保在给定的时间里，只有一个用户能执行写入，并防止其他用户读取正在写入的同一资源。

在实际的数据库系统中，每时每刻都在发生锁定，当某个用户在修改某一部分数据时，MySQL 会通过锁定防止其他用户读取同一数据。大多数时候，MySQL 锁的内部管理都是透明的。

1.2.2 锁粒度

一种提高共享资源并发性的方式就是让锁定对象更有选择性。尽量只锁定需要修改的部
5 分数据，而不是所有的资源。更理想的方式是，只对会修改的数据片进行精确的锁定。任何时候，在给定的资源上，锁定的数据量越少，则系统的并发程度越高，只要相互之间不发生冲突即可。

问题是加锁也需要消耗资源。锁的各种操作，包括获得锁、检查锁是否已经解除、释放锁等，都会增加系统的开销。如果系统花费大量的时间来管理锁，而不是存取数据，那

么系统的性能可能会因此受到影响。

所谓的锁策略，就是在锁的开销和数据的安全性之间寻求平衡，这种平衡当然也会影响到性能。大多数商业数据库系统没有提供更多的选择，一般都是在表上施加行级锁（row-level lock），并以各种复杂的方式来实现，以便在锁比较多的情况下尽可能地提供更好的性能。

而MySQL则提供了多种选择。每种MySQL存储引擎都可以实现自己的锁策略和锁粒度。在存储引擎的设计中，锁管理是个非常重要的决定。将锁粒度固定在某个级别，可以为某些特定的应用场景提供更好的性能，但同时却会失去对另外一些应用场景的良好支持。好在MySQL支持多个存储引擎的架构，所以不需要单一的通用解决方案。下面将介绍两种最重要的锁策略。

表锁（table lock）

表锁是MySQL中最基本的锁策略，并且是开销最小的策略。表锁非常类似于前文描述的邮箱加锁机制：它会锁定整张表。一个用户在对表进行写操作（插入、删除、更新等）前，需要先获得写锁，这会阻塞其他用户对该表的所有读写操作。只有没有写锁时，其他读取的用户才能获得读锁，读锁之间是不相互阻塞的。

在特定的场景中，表锁也可能有良好的性能。例如，READ LOCAL表锁支持某些类型的并发写操作。另外，写锁也比读锁有更高的优先级，因此一个写锁请求可能会被插入到读锁队列的前面（写锁可以插入到锁队列中读锁的前面，反之读锁则不能插入到写锁的前面）。

尽管存储引擎可以管理自己的锁，MySQL本身还是会使用各种有效的表锁来实现不同的目的。例如，服务器会为诸如ALTER TABLE之类的语句使用表锁，而忽略存储引擎的锁机制。

行级锁（row lock）

行级锁可以最大程度地支持并发处理（同时也带来了最大的锁开销）。众所周知，在InnoDB和XtraDB，以及其他一些存储引擎中实现了行级锁。行级锁只在存储引擎层实现，而MySQL服务器层（如有必要，请回顾前文的逻辑架构图）没有实现。服务器层完全不了解存储引擎中的锁实现。在本章的后续内容以及全书中，所有的存储引擎都以自己的方式显现了锁机制。 6

1.3 事务

在理解事务的概念之前，接触数据库系统的其他高级特性还言之过早。事务就是一组原子性的 SQL 查询，或者说一个独立的工作单元。如果数据库引擎能够成功地对数据库应用该组查询的全部语句，那么就执行该组查询。如果其中有任何一条语句因为崩溃或其他原因无法执行，那么所有的语句都不会执行。也就是说，事务内的语句，要么全部执行成功，要么全部执行失败。

本节的内容并非专属于 MySQL，如果读者已经熟悉了事务的 ACID 的概念，可以直接跳转到 1.3.4 节。

银行应用是解释事务必要性的一个经典例子。假设一个银行的数据库有两张表：支票（checking）表和储蓄（savings）表。现在要从用户 Jane 的支票账户转移 200 美元到她的储蓄账户，那么需要至少三个步骤：

1. 检查支票账户的余额高于 200 美元。
2. 从支票账户余额中减去 200 美元。
3. 在储蓄账户余额中增加 200 美元。

上述三个步骤的操作必须打包在一个事务中，任何一个步骤失败，则必须回滚所有的步骤。

可以用 START TRANSACTION 语句开始一个事务，然后要么使用 COMMIT 提交事务将修改的数据持久保留，要么使用 ROLLBACK 撤销所有的修改。事务 SQL 的样本如下：

```
1  START TRANSACTION;
2  SELECT balance FROM checking WHERE customer_id = 10233276;
3  UPDATE checking SET balance = balance - 200.00 WHERE customer_id = 10233276;
4  UPDATE savings  SET balance = balance + 200.00 WHERE customer_id = 10233276;
5  COMMIT;
```

单纯的事务概念并不是故事的全部。试想一下，如果执行到第四条语句时服务器崩溃了，会发生什么？天知道，用户可能会损失 200 美元。再假如，在执行到第三条语句和第四条语句之间时，另外一个进程要删除支票账户的所有余额，那么结果可能就是银行在不知道这个逻辑的情况下白白给了 Jane 200 美元。

除非系统通过严格的 ACID 测试，否则空谈事务的概念是不够的。ACID 表示原子性（atomicity）、一致性（consistency）、隔离性（isolation）和持久性（durability）。一个运行良好的事务处理系统，必须具备这些标准特征。

原子性（atomicity）

一个事务必须被视为一个不可分割的最小工作单元，整个事务中的所有操作要么全部提交成功，要么全部失败回滚，对于一个事务来说，不可能只执行其中的一部分操作，这就是事务的原子性。

一致性（consistency） 7

数据库总是从一个一致性的状态转换到另外一个一致性的状态。在前面的例子中，一致性确保了，即使在执行第三、四条语句之间时系统崩溃，支票账户中也不会损失 200 美元，因为事务最终没有提交，所以事务中所做的修改也不会保存到数据库中。

隔离性（isolation）

通常来说，一个事务所做的修改在最终提交以前，对其他事务是不可见的。在前面的例子中，当执行完第三条语句、第四条语句还未开始时，此时有另外一个账户汇总程序开始运行，则其看到的支票账户的余额并没有被减去 200 美元。后面我们讨论隔离级别（Isolation level）的时候，会发现为什么我们要说“通常来说”是不可见的。

持久性（durability）

一旦事务提交，则其所做的修改就会永久保存到数据库中。此时即使系统崩溃，修改的数据也不会丢失。持久性是个有点模糊的概念，因为实际上持久性也分很多不同的级别。有些持久性策略能够提供非常强的安全保障，而有些则未必。而且不可能有能做到 100% 的持久性保证的策略（如果数据库本身就能做到真正的持久性，那么备份又怎么能增加持久性呢？）。在后面的一些章节中，我们会继续讨论 MySQL 中持久性的真正含义。

事务的 ACID 特性可以确保银行不会弄丢你的钱。而在应用逻辑中，要实现这一点非常难，甚至可以说是不可能完成的任务。一个兼容 ACID 的数据库系统，需要做很多复杂但可能用户并没有觉察到的工作，才能确保 ACID 的实现。

就像锁粒度的升级会增加系统开销一样，这种事务处理过程中额外的安全性，也会需要数据库系统做更多的额外工作。一个实现了 ACID 的数据库，相比没有实现 ACID 的数据库，通常会需要更强的 CPU 处理能力、更大的内存和更多的磁盘空间。正如本章不断重复的，这也正是 MySQL 的存储引擎架构可以发挥优势的地方。用户可以根据业务是否需要事务处理，来选择合适的存储引擎。对于一些不需要事务的查询类应用，选择一个非事务型的存储引擎，可以获得更高的性能。即使存储引擎不支持事务，也可以通过 LOCK TABLES 语句为应用提供一定程度的保护，这些选择用户都可以自主决定。

1.3.1 隔离级别

隔离性其实比想象的要复杂。在 SQL 标准中定义了四种隔离级别，每一种级别都规定了一个事务中所做的修改，哪些在事务内和事务间是可见的，哪些是不可见的。较低级别的隔离通常可以执行更高的并发，系统的开销也更低。

8 每种存储引擎实现的隔离级别不尽相同。如果熟悉其他的数据库产品，可能会发现某些特性和你期望的会有些不一样（但本节不打算讨论更详细的内容）。读者可以根据所选择的存储引擎，查阅相关的手册。

下面简单地介绍一下四种隔离级别。

READ UNCOMMITTED（未提交读）

在 READ UNCOMMITTED 级别，事务中的修改，即使没有提交，对其他事务也都是可见的。事务可以读取未提交的数据，这也被称为脏读（Dirty Read）。这个级别会导致很多问题，从性能上来说，READ UNCOMMITTED 不会比其他的级别好太多，但却缺乏其他级别的很多好处，除非真的有非常必要的理由，在实际应用中一般很少使用。

READ COMMITTED（提交读）

大多数数据库系统的默认隔离级别都是 READ COMMITTED（但 MySQL 不是）。READ COMMITTED 满足前面提到的隔离性的简单定义：一个事务开始时，只能“看见”已经提交的事务所做的修改。换句话说，一个事务从开始直到提交之前，所做的任何修改对其他事务都是不可见的。这个级别有时候也叫做不可重复读（nonrepeatable read），因为两次执行同样的查询，可能会得到不一样的结果。

REPEATABLE READ（可重复读）

REPEATABLE READ 解决了脏读的问题。该级别保证了在同一个事务中多次读取同样记录的结果是一致的。但是理论上，可重复读隔离级别还是无法解决另外一个幻读（Phantom Read）的问题。所谓幻读，指的是当某个事务在读取某个范围内的记录时，另外一个事务又在该范围内插入了新的记录，当之前的事务再次读取该范围的记录时，会产生幻行（Phantom Row）。InnoDB 和 XtraDB 存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）解决了幻读的问题。本章稍后会做进一步的讨论。

可重复读是 MySQL 的默认事务隔离级别。

SERIALIZABLE（可串行化）

SERIALIZABLE 是最高的隔离级别。它通过强制事务串行执行，避免了前面说的幻读的问题。简单来说，SERIALIZABLE 会在读取的每一行数据上都加锁，所以可能导致大量的超时和锁争用的问题。实际应用中也很少用到这个隔离级别，只有在非常需要确保数据的一致性而且可以接受没有并发的情况下，才考虑采用该级别。

表1-1：ANSI SQL隔离级别

隔离级别	脏读可能性	不可重复读可能性	幻读可能性	加锁读
READ UNCOMMITTED	Yes	Yes	Yes	No
READ COMMITTED	No	Yes	Yes	No
REPEATABLE READ	No	No	Yes	No
SERIALIZABLE	No	No	No	Yes

1.3.2 死锁

死锁是指两个或者多个事务在同一资源上相互占用，并请求锁定对方占用的资源，从而导致恶性循环的现象。当多个事务试图以不同的顺序锁定资源时，就可能会产生死锁。多个事务同时锁定同一个资源时，也会产生死锁。例如，设想下面两个事务同时处理 StockPrice 表：

事务 1

```
START TRANSACTION;
UPDATE StockPrice SET close = 45.50 WHERE stock_id = 4 and date = '2002-05-01';
UPDATE StockPrice SET close = 19.80 WHERE stock_id = 3 and date = '2002-05-02';
COMMIT;
```

事务 2

```
START TRANSACTION;
UPDATE StockPrice SET high  = 20.12 WHERE stock_id = 3 and date = '2002-05-02';
UPDATE StockPrice SET high  = 47.20 WHERE stock_id = 4 and date = '2002-05-01';
COMMIT;
```

如果凑巧，两个事务都执行了第一条 UPDATE 语句，更新了一行数据，同时也锁定了该行数据，接着每个事务都尝试去执行第二条 UPDATE 语句，却发现该行已经被对方锁定，然后两个事务都等待对方释放锁，同时又持有对方需要的锁，则陷入死循环。除非有外部因素介入才可能解除死锁。

为了解决这种问题，数据库系统实现了各种死锁检测和死锁超时机制。越复杂的系统，比如 InnoDB 存储引擎，越能检测到死锁的循环依赖，并立即返回一个错误。这种解决方式很有效，否则死锁会导致出现非常慢的查询。还有一种解决方式，就是当查询的时间达到锁等待超时的设定后放弃锁请求，这种方式通常来说不太好。InnoDB 目前处理死锁的方法是，将持有最少行级排他锁的事务进行回滚（这是相对比较简单的死锁回滚算法）。

锁的行为和顺序是和存储引擎相关的。以同样的顺序执行语句，有些存储引擎会产生死锁，有些则不会。死锁的产生有双重原因：有些是因为真正的数据冲突，这种情况通常很难避免，但有些则完全是由于存储引擎的实现方式导致的。

10 死锁发生以后，只有部分或者完全回滚其中一个事务，才能打破死锁。对于事务型的系统，这是无法避免的，所以应用程序在设计时必须考虑如何处理死锁。大多数情况下只需要重新执行因死锁回滚的事务即可。

1.3.3 事务日志

事务日志可以帮助提高事务的效率。使用事务日志，存储引擎在修改表的数据时只需要修改其内存拷贝，再把该修改行为记录到持久在硬盘上的事务日志中，而不用每次都将修改的数据本身持久到磁盘。事务日志采用的是追加的方式，因此写日志的操作是磁盘上一小块区域内的顺序 I/O，而不像随机 I/O 需要在磁盘的多个地方移动磁头，所以采用事务日志的方式相对来说要快得多。事务日志持久以后，内存中被修改的数据在后台可以慢慢地刷回到磁盘。目前大多数存储引擎都是这样实现的，我们通常称之为预写式日志（Write-Ahead Logging），修改数据需要写两次磁盘。

如果数据的修改已经记录到事务日志并持久化，但数据本身还没有写回磁盘，此时系统崩溃，存储引擎在重启时能够自动恢复这部分修改的数据。具体的恢复方式则视存储引擎而定。

1.3.4 MySQL 中的事务

MySQL 提供了两种事务型的存储引擎：InnoDB 和 NDB Cluster。另外还有一些第三方存储引擎也支持事务，比较知名的包括 XtraDB 和 PBXT。后面将详细讨论它们各自的一些特点。

自动提交（AUTOCOMMIT）

MySQL 默认采用自动提交（`AUTOCOMMIT`）模式。也就是说，如果不是显式地开始一个事务，则每个查询都被当作一个事务执行提交操作。在当前连接中，可以通过设置 `AUTOCOMMIT` 变量来启用或者禁用自动提交模式：

```
mysql> SHOW VARIABLES LIKE 'AUTOCOMMIT';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.00 sec)
mysql> SET AUTOCOMMIT = 1;
```

`1` 或者 `ON` 表示启用，`0` 或者 `OFF` 表示禁用。当 `AUTOCOMMIT=0` 时，所有的查询都是在一个事务中，直到显式地执行 `COMMIT` 提交或者 `ROLLBACK` 回滚，该事务结束，同时又开始了

另一个新事务。修改 AUTOCOMMIT 对非事务型的表，比如 MyISAM 或者内存表，不会有任何影响。对这类表来说，没有 COMMIT 或者 ROLLBACK 的概念，也可以说是相当于一直处于 AUTOCOMMIT 启用的模式。

另外还有一些命令，在执行之前会强制执行 COMMIT 提交当前的活动事务。典型的例子，11 在数据定义语言（DDL）中，如果是会导致大量数据改变的操作，比如 ALTER TABLE，就是如此。另外还有 LOCK TABLES 等其他语句也会导致同样的结果。如果有需要，请检查对应版本的官方文档来确认所有可能导致自动提交的语句列表。

MySQL 可以通过执行 SET TRANSACTION ISOLATION LEVEL 命令来设置隔离级别。新的隔离级别会在下一个事务开始的时候生效。可以在配置文件中设置整个数据库的隔离级别，也可以只改变当前会话的隔离级别：

```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

MySQL 能够识别所有的 4 个 ANSI 隔离级别，InnoDB 引擎也支持所有的隔离级别。

在事务中混合使用存储引擎

MySQL 服务器层不管理事务，事务是由下层的存储引擎实现的。所以在同一个事务中，使用多种存储引擎是不可靠的。

如果在事务中混合使用了事务型和非事务型的表（例如 InnoDB 和 MyISAM 表），在正常提交的情况下不会有什么问题。

但如果该事务需要回滚，非事务型的表上的变更就无法撤销，这会导致数据库处于不一致的状态，这种情况很难修复，事务的最终结果将无法确定。所以，为每张表选择合适的存储引擎非常重要。

在非事务型的表上执行事务相关操作的时候，MySQL 通常不会发出提醒，也不会报错。有时候只有回滚的时候才会发出一个警告："某些非事务型的表上的变更不能被回滚"。但大多数情况下，对非事务型表的操作都不会有提示。

隐式和显式锁定

InnoDB 采用的是两阶段锁定协议（two-phase locking protocol）。在事务执行过程中，随时都可以执行锁定，锁只有在执行 COMMIT 或者 ROLLBACK 的时候才会释放，并且所有的锁是在同一时刻被释放。前面描述的锁定都是隐式锁定，InnoDB 会根据隔离级别在需要的时候自动加锁。

另外，InnoDB 也支持通过特定的语句进行显式锁定，这些语句不属于 SQL 规范[注3]：

- SELECT ... LOCK IN SHARE MODE
- SELECT ... FOR UPDATE

12 MySQL 也支持 LOCK TABLES 和 UNLOCK TABLES 语句，这是在服务器层实现的，和存储引擎无关。它们有自己的用途，但并不能替代事务处理。如果应用需要用到事务，还是应该选择事务型存储引擎。

经常可以发现，应用已经将表从 MyISAM 转换到 InnoDB，但还是显式地使用 LOCK TABLES 语句。这不但没有必要，还会严重影响性能，实际上 InnoDB 的行级锁工作得更好。

> LOCK TABLES 和事务之间相互影响的话，情况会变得非常复杂，在某些 MySQL 版本中甚至会产生无法预料的结果。因此，本书建议，除了事务中禁用了 AUTOCOMMIT，可以使用 LOCK TABLES 之外，其他任何时候都不要显式地执行 LOCK TABLES，不管使用的是什么存储引擎。

1.4 多版本并发控制

MySQL 的大多数事务型存储引擎实现的都不是简单的行级锁。基于提升并发性能的考虑，它们一般都同时实现了多版本并发控制（MVCC）。不仅是 MySQL，包括 Oracle、PostgreSQL 等其他数据库系统也都实现了 MVCC，但各自的实现机制不尽相同，因为 MVCC 没有一个统一的实现标准。

可以认为 MVCC 是行级锁的一个变种，但是它在很多情况下避免了加锁操作，因此开销更低。虽然实现机制有所不同，但大都实现了非阻塞的读操作，写操作也只锁定必要的行。

MVCC 的实现，是通过保存数据在某个时间点的快照来实现的。也就是说，不管需要执行多长时间，每个事务看到的数据都是一致的。根据事务开始的时间不同，每个事务对同一张表，同一时刻看到的数据可能是不一样的。如果之前没有这方面的概念，这句话听起来就有点迷惑。熟悉了以后会发现，这句话其实还是很容易理解的。

前面说到不同存储引擎的 MVCC 实现是不同的，典型的有乐观（optimistic）并发控制和悲观（pessimistic）并发控制。下面我们通过 InnoDB 的简化版行为来说明 MVCC 是如何工作的。

注 3： 这些锁定提示经常被滥用，实际上应当尽量避免使用。第 6 章有更详细的讨论。

InnoDB 的 MVCC，是通过在每行记录后面保存两个隐藏的列来实现的。这两个列，一个保存了行的创建时间，一个保存行的过期时间（或删除时间）。当然存储的并不是实际的时间值，而是系统版本号（system version number）。每开始一个新的事务，系统版本号都会自动递增。事务开始时刻的系统版本号会作为事务的版本号，用来和查询到的每行记录的版本号进行比较。下面看一下在 REPEATABLE READ 隔离级别下，MVCC 具体是如何操作的。 13

SELECT

InnoDB 会根据以下两个条件检查每行记录：

a. InnoDB 只查找版本早于当前事务版本的数据行（也就是，行的系统版本号小于或等于事务的系统版本号），这样可以确保事务读取的行，要么是在事务开始前已经存在的，要么是事务自身插入或者修改过的。

b. 行的删除版本要么未定义，要么大于当前事务版本号。这可以确保事务读取到的行，在事务开始之前未被删除。

只有符合上述两个条件的记录，才能返回作为查询结果。

INSERT

InnoDB 为新插入的每一行保存当前系统版本号作为行版本号。

DELETE

InnoDB 为删除的每一行保存当前系统版本号作为行删除标识。

UPDATE

InnoDB 为插入一行新记录，保存当前系统版本号作为行版本号，同时保存当前系统版本号到原来的行作为行删除标识。

保存这两个额外系统版本号，使大多数读操作都可以不用加锁。这样设计使得读数据操作很简单，性能很好，并且也能保证只会读取到符合标准的行。不足之处是每行记录都需要额外的存储空间，需要做更多的行检查工作，以及一些额外的维护工作。

MVCC 只在 REPEATABLE READ 和 READ COMMITTED 两个隔离级别下工作。其他两个隔离级别都和 MVCC 不兼容[注4]，因为 READ UNCOMMITTED 总是读取最新的数据行，而不是符合当前事务版本的数据行。而 SERIALIZABLE 则会对所有读取的行都加锁。

1.5 MySQL 的存储引擎

本节只是概要地描述 MySQL 的存储引擎，而不会涉及太多细节。因为关于存储引擎的讨论及其相关特性将会贯穿全书，而且本书也不是存储引擎的完全指南，所以有必要阅

注 4： MVCC 并没有正式的规范，所以各个存储引擎和数据库系统的实现都是各异的，没有人能说其他的实现方式是错误的。

读相关存储引擎的官方文档。

在文件系统中，MySQL 将每个数据库（也可以称之为 schema）保存为数据目录下的一
14 个子目录。创建表时，MySQL 会在数据库子目录下创建一个和表同名的 *.frm* 文件保存表的定义。例如创建一个名为 MyTable 的表，MySQL 会在 *MyTable.frm* 文件中保存该表的定义。因为 MySQL 使用文件系统的目录和文件来保存数据库和表的定义，大小写敏感性和具体的平台密切相关。在 Windows 中，大小写是不敏感的；而在类 Unix 中则是敏感的。不同的存储引擎保存数据和索引的方式是不同的，但表的定义则是在 MySQL 服务层统一处理的。

可以使用 SHOW TABLE STATUS 命令（在 MySQL 5.0 以后的版本中，也可以查询 INFORMATION_SCHEMA 中对应的表）显示表的相关信息。例如，对于 mysql 数据库中的 user 表：

```
mysql> SHOW TABLE STATUS LIKE 'user' \G
*************************** 1. row ***************************
           Name: user
         Engine: MyISAM
     Row_format: Dynamic
           Rows: 6
 Avg_row_length: 59
    Data_length: 356
Max_data_length: 4294967295
   Index_length: 2048
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2002-01-24 18:07:17
    Update_time: 2002-01-24 21:56:29
     Check_time: NULL
      Collation: utf8_bin
       Checksum: NULL
 Create_options:
        Comment: Users and global privileges
1 row in set (0.00 sec)
```

输出的结果表明，这是一个 MyISAM 表。输出中还有很多其他信息以及统计信息。下面简单介绍一下每一行的含义。

Name

　　表名。

Engine

　　表的存储引擎类型。在旧版本中，该列的名字叫 Type，而不是 Engine。

Row_format

　　行的格式。对于 MyISAM 表，可选的值为 Dynamic、Fixed 或者 Compressed。Dynamic 的行长度是可变的，一般包含可变长度的字段，如 VARCHAR 或 BLOB。Fixed

的行长度则是固定的，只包含固定长度的列，如 CHAR 和 INTEGER。Compressed 的行则只在压缩表中存在，请参考第 19 页“MyISAM 压缩表”一节。

Rows

表中的行数。对于 MyISAM 和其他一些存储引擎，该值是精确的，但对于 InnoDB，该值是估计值。

Avg_row_length 15

平均每行包含的字节数。

Data_length

表数据的大小（以字节为单位）。

Max_data_length

表数据的最大容量，该值和存储引擎有关。

Index_length

索引的大小（以字节为单位）。

Data_free

对于 MyISAM 表，表示已分配但目前没有使用的空间。这部分空间包括了之前删除的行，以及后续可以被 INSERT 利用到的空间。

Auto_increment

下一个 AUTO_INCREMENT 的值。

Create_time

表的创建时间。

Update_time

表数据的最后修改时间。

Check_time

使用 CKECK TABLE 命令或者 *myisamchk* 工具最后一次检查表的时间。

Collation

表的默认字符集和字符列排序规则。

Checksum

如果启用，保存的是整个表的实时校验和。

Create_options

创建表时指定的其他选项。

Comment

该列包含了一些其他的额外信息。对于 MyISAM 表，保存的是表在创建时带的注释。对于 InnoDB 表，则保存的是 InnoDB 表空间的剩余空间信息。如果是一个视图，则该列包含“VIEW”的文本字样。

1.5.1 InnoDB 存储引擎

InnoDB 是 MySQL 的默认事务型引擎，也是最重要、使用最广泛的存储引擎。它被设计用来处理大量的短期（short-lived）事务，短期事务大部分情况是正常提交的，很少会被回滚。InnoDB 的性能和自动崩溃恢复特性，使得它在非事务型存储的需求中也很流行。除非有非常特别的原因需要使用其他的存储引擎，否则应该优先考虑 InnoDB 引擎。
16 如果要学习存储引擎，InnoDB 也是一个非常好的值得花最多的时间去深入学习的对象，收益肯定比将时间平均花在每个存储引擎的学习上要高得多。

InnoDB 的历史

InnoDB 有着复杂的发布历史，了解一下这段历史对于理解 InnoDB 很有帮助。2008 年，发布了所谓的 InnoDB plugin，适用于 MySQL 5.1 版本，但这是 Oracle 创建的下一代 InnoDB 引擎，其拥有者是 InnoDB 而不是 MySQL。这基于很多原因，这些原因如果要一一道来，恐怕得喝掉好几桶啤酒。MySQL 默认还是选择了集成旧的 InnoDB 引擎。当然用户可以自行选择使用新的性能更好、扩展性更佳的 InnoDB plugin 来覆盖旧的版本。直到最后，在 Oracle 收购了 Sun 公司后发布的 MySQL 5.5 中才彻底使用 InnoDB plugin 替代了旧版本的 InnoDB（是的，这也意味着 InnoDB plugin 已经是原生编译了，而不是编译成一个插件，但名字已经约定俗成很难更改）。

这个现代的 InnoDB 版本，也就是 MySQL 5.1 中所谓的 InnoDB plugin，支持一些新特性，诸如利用排序创建索引（building index by sorting）、删除或者增加索引时不需要复制全表数据、新的支持压缩的存储格式、新的大型列值如 BLOB 的存储方式，以及文件格式管理等。很多用户在 MySQL 5.1 中没有使用 InnoDB plugin，或许是因为他们没有注意到有这个区别。所以如果你使用的是 MySQL 5.1，一定要使用 InnoDB plugin，真的比旧版本的 InnoDB 要好很多。

InnoDB 是一个很重要的存储引擎，很多个人和公司都对其贡献代码，而不仅仅是 Oracle 公司的开发团队。一些重要的贡献者包括 Google、Yasufumi Kinoshita、Percona、Facebook 等，他们的一些改进被直接移植到官方版本，也有一些由 InnoDB 团队重新实现。在过去的几年间，InnoDB 的改进速度大大加快，主要的改进集中在可测量性、可扩展性、可配置化、性能、各种新特性和对 Windows 的支持等方面。MySQL 5.6 实验室预览版和里程碑版也包含了一系列重要的 InnoDB 新特性。

为改善 InnoDB 的性能，Oracle 投入了大量的资源，并做了很多卓有成效的工作（外部贡献者对此也提供了很大的帮助）。在本书的第二版中，我们注意到在超过四核 CPU 的系统中 InnoDB 表现不佳，而现在已经可以很好地扩展至 24 核的系统，甚至在某些场景，32 核或者更多核的系统中也表现良好。很多改进将在即将发布的 MySQL 5.6 中引入，

当然也还有机会做更进一步的改善。

InnoDB 概览

InnoDB 的数据存储在表空间（tablespace）中，表空间是由 InnoDB 管理的一个黑盒子，由一系列的数据文件组成。在 MySQL 4.1 以后的版本中，InnoDB 可以将每个表的数据和索引存放在单独的文件中。InnoDB 也可以使用裸设备作为表空间的存储介质，但现 17
代的文件系统使得裸设备不再是必要的选择。

InnoDB 采用 MVCC 来支持高并发，并且实现了四个标准的隔离级别。其默认级别是 REPEATABLE READ（可重复读），并且通过间隙锁（next-key locking）策略防止幻读的出现。间隙锁使得 InnoDB 不仅仅锁定查询涉及的行，还会对索引中的间隙进行锁定，以防止幻影行的插入。

InnoDB 表是基于聚簇索引建立的，我们会在后面的章节详细讨论聚簇索引。InnoDB 的索引结构和 MySQL 的其他存储引擎有很大的不同，聚簇索引对主键查询有很高的性能。不过它的二级索引（secondary index，非主键索引）中必须包含主键列，所以如果主键列很大的话，其他的所有索引都会很大。因此，若表上的索引较多的话，主键应当尽可能的小。InnoDB 的存储格式是平台独立的，也就是说可以将数据和索引文件从 Intel 平台复制到 PowerPC 或者 Sun SPARC 平台。

InnoDB 内部做了很多优化，包括从磁盘读取数据时采用的可预测性预读，能够自动在内存中创建 hash 索引以加速读操作的自适应哈希索引（adaptive hash index），以及能够加速插入操作的插入缓冲区（insert buffer）等。本书后面将更详细地讨论这些内容。

InnoDB 的行为是非常复杂的，不容易理解。如果使用了 InnoDB 引擎，笔者强烈建议阅读官方手册中的"InnoDB 事务模型和锁"一节。如果应用程序基于 InnoDB 构建，则事先了解一下 InnoDB 的 MVCC 架构带来的一些微妙和细节之处是非常有必要的。存储引擎要为所有用户甚至包括修改数据的用户维持一致性的视图，是非常复杂的工作。

作为事务型的存储引擎，InnoDB 通过一些机制和工具支持真正的热备份，Oracle 提供的 MySQL Enterprise Backup、Percona 提供的开源的 XtraBackup 都可以做到这一点。MySQL 的其他存储引擎不支持热备份，要获取一致性视图需要停止对所有表的写入，而在读写混合场景中，停止写入可能也意味着停止读取。

1.5.2 MyISAM 存储引擎

在 MySQL 5.1 及之前的版本，MyISAM 是默认的存储引擎。MyISAM 提供了大量的特性，包括全文索引、压缩、空间函数（GIS）等，但 MyISAM 不支持事务和行级锁，而

且有一个毫无疑问的缺陷就是崩溃后无法安全恢复。正是由于 MyISAM 引擎的缘故，即使 MySQL 支持事务已经很长时间了，在很多人的概念中 MySQL 还是非事务型的数据库。尽管 MyISAM 引擎不支持事务、不支持崩溃后的安全恢复，但它绝不是一无是处
18 的。对于只读的数据，或者表比较小、可以忍受修复（repair）操作，则依然可以继续使用 MyISAM（但请不要默认使用 MyISAM，而是应当默认使用 InnoDB）。

存储

MyISAM 会将表存储在两个文件中：数据文件和索引文件，分别以 *.MYD* 和 *.MYI* 为扩展名。MyISAM 表可以包含动态或者静态（长度固定）行。MySQL 会根据表的定义来决定采用何种行格式。MyISAM 表可以存储的行记录数，一般受限于可用的磁盘空间，或者操作系统中单个文件的最大尺寸。

在 MySQL 5.0 中，MyISAM 表如果是变长行，则默认配置只能处理 256TB 的数据，因为指向数据记录的指针长度是 6 个字节。而在更早的版本中，指针长度默认是 4 字节，所以只能处理 4GB 的数据。而所有的 MySQL 版本都支持 8 字节的指针。要改变 MyISAM 表指针的长度（调高或者调低），可以通过修改表的 `MAX_ROWS` 和 `AVG_ROW_LENGTH` 选项的值来实现，两者相乘就是表可能达到的最大大小。修改这两个参数会导致重建整个表和表的所有索引，这可能需要很长的时间才能完成。

MyISAM 特性

作为 MySQL 最早的存储引擎之一，MyISAM 有一些已经开发出来很多年的特性，可以满足用户的实际需求。

加锁与并发

MyISAM 对整张表加锁，而不是针对行。读取时会对需要读到的所有表加共享锁，写入时则对表加排他锁。但是在表有读取查询的同时，也可以往表中插入新的记录（这被称为并发插入，CONCURRENT INSERT）。

修复

对于 MyISAM 表，MySQL 可以手工或者自动执行检查和修复操作，但这里说的修复和事务恢复以及崩溃恢复是不同的概念。执行表的修复可能导致一些数据丢失，而且修复操作是非常慢的。可以通过 `CHECK TABLE mytable` 检查表的错误，如果有错误可以通过执行 `REPAIR TABLE mytable` 进行修复。另外，如果 MySQL 服务器已经关闭，也可以通过 *myisamchk* 命令行工具进行检查和修复操作。

索引特性

对于 MyISAM 表，即使是 `BLOB` 和 `TEXT` 等长字段，也可以基于其前 500 个字符创建

索引。MyISAM 也支持全文索引，这是一种基于分词创建的索引，可以支持复杂的查询。关于索引的更多信息请参考第 5 章。

延迟更新索引键（Delayed Key Write）

创建 MyISAM 表的时候，如果指定了 DELAY_KEY_WRITE 选项，在每次修改执行完成时，不会立刻将修改的索引数据写入磁盘，而是会写到内存中的键缓冲区（in-memory key buffer），只有在清理键缓冲区或者关闭表的时候才会将对应的索引块写入到磁盘。这种方式可以极大地提升写入性能，但是在数据库或者主机崩溃时会造成索引损坏，需要执行修复操作。延迟更新索引键的特性，可以在全局设置，也可以为单个表设置。

MyISAM 压缩表

如果表在创建并导入数据以后，不会再进行修改操作，那么这样的表或许适合采用 MyISAM 压缩表。

可以使用 *myisampack* 对 MyISAM 表进行压缩（也叫打包 pack）。压缩表是不能进行修改的（除非先将表解除压缩，修改数据，然后再次压缩）。压缩表可以极大地减少磁盘空间占用，因此也可以减少磁盘 I/O，从而提升查询性能。压缩表也支持索引，但索引也是只读的。

以现在的硬件能力，对大多数应用场景，读取压缩表数据时的解压带来的开销影响并不大，而减少 I/O 带来的好处则要大得多。压缩时表中的记录是独立压缩的，所以读取单行的时候不需要去解压整个表（甚至也不解压行所在的整个页面）。

MyISAM 性能

MyISAM 引擎设计简单，数据以紧密格式存储，所以在某些场景下的性能很好。MyISAM 有一些服务器级别的性能扩展限制，比如对索引键缓冲区（key cache）的 Mutex 锁，MariaDB 基于段（segment）的索引键缓冲区机制来避免该问题。但 MyISAM 最典型的性能问题还是表锁的问题，如果你发现所有的查询都长期处于“Locked”状态，那么毫无疑问表锁就是罪魁祸首。

1.5.3 MySQL 内建的其他存储引擎

MySQL 还有一些有特殊用途的存储引擎。在新版本中，有些可能因为一些原因已经不再支持；另外还有些会继续支持，但是需要明确地启用后才能使用。

Archive 引擎

Archive 存储引擎只支持 INSERT 和 SELECT 操作，在 MySQL 5.1 之前也不支持索引。

Archive 引擎会缓存所有的写并利用 *zlib* 对插入的行进行压缩，所以比 MyISAM 表的磁盘 I/O 更少。但是每次 SELECT 查询都需要执行全表扫描。所以 Archive 表适合日志和数据采集类应用，这类应用做数据分析时往往需要全表扫描。或者在一些需要更快速的 INSERT 操作的场合下也可以使用。

20 Archive 引擎支持行级锁和专用的缓冲区，所以可以实现高并发的插入。在一个查询开始直到返回表中存在的所有行数之前，Archive 引擎会阻止其他的 SELECT 执行，以实现一致性读。另外，也实现了批量插入在完成之前对读操作是不可见的。这种机制模仿了事务和 MVCC 的一些特性，但 Archive 引擎不是一个事务型的引擎，而是一个针对高速插入和压缩做了优化的简单引擎。

Blackhole 引擎

Blackhole 引擎没有实现任何的存储机制，它会丢弃所有插入的数据，不做任何保存。但是服务器会记录 Blackhole 表的日志，所以可以用于复制数据到备库，或者只是简单地记录到日志。这种特殊的存储引擎可以在一些特殊的复制架构和日志审核时发挥作用。但这种应用方式我们碰到过很多问题，因此并不推荐。

CSV 引擎

CSV 引擎可以将普通的 CSV 文件（逗号分割值的文件）作为 MySQL 的表来处理，但这种表不支持索引。CSV 引擎可以在数据库运行时拷入或者拷出文件。可以将 Excel 等电子表格软件中的数据存储为 CSV 文件，然后复制到 MySQL 数据目录下，就能在 MySQL 中打开使用。同样，如果将数据写入到一个 CSV 引擎表，其他的外部程序也能立即从表的数据文件中读取 CSV 格式的数据。因此 CSV 引擎可以作为一种数据交换的机制，非常有用。

Federated 引擎

Federated 引擎是访问其他 MySQL 服务器的一个代理，它会创建一个到远程 MySQL 服务器的客户端连接，并将查询传输到远程服务器执行，然后提取或者发送需要的数据。最初设计该存储引擎是为了和企业级数据库如 Microsoft SQL Server 和 Oracle 的类似特性竞争的，可以说更多的是一种市场行为。尽管该引擎看起来提供了一种很好的跨服务器的灵活性，但也经常带来问题，因此默认是禁用的。MariaDB 使用了它的一个后续改进版本，叫做 FederatedX。

Memory 引擎

如果需要快速地访问数据，并且这些数据不会被修改，重启以后丢失也没有关系，那么

使用 Memory 表（以前也叫做 HEAP 表）是非常有用的。Memory 表至少比 MyISAM 表要快一个数量级，因为所有的数据都保存在内存中，不需要进行磁盘 I/O。Memory 表的结构在重启以后还会保留，但数据会丢失。

Memroy 表在很多场景可以发挥好的作用：

- 用于查找（lookup）或者映射（mapping）表，例如将邮编和州名映射的表。
- 用于缓存周期性聚合数据（periodically aggregated data）的结果。 21
- 用于保存数据分析中产生的中间数据。

Memory 表支持 Hash 索引，因此查找操作非常快。虽然 Memory 表的速度非常快，但还是无法取代传统的基于磁盘的表。Memroy 表是表级锁，因此并发写入的性能较低。它不支持 BLOB 或 TEXT 类型的列，并且每行的长度是固定的，所以即使指定了 VARCHAR 列，实际存储时也会转换成 CHAR，这可能导致部分内存的浪费（其中一些限制在 Percona 版本已经解决）。

如果 MySQL 在执行查询的过程中需要使用临时表来保存中间结果，内部使用的临时表就是 Memory 表。如果中间结果太大超出了 Memory 表的限制，或者含有 BLOB 或 TEXT 字段，则临时表会转换成 MyISAM 表。在后续的章节还会继续讨论该问题。

> 人们经常混淆 Memory 表和临时表。临时表是指使用 CREATE TEMPORARY TABLE 语句创建的表，它可以使用任何存储引擎，因此和 Memory 表不是一回事。临时表只在单个连接中可见，当连接断开时，临时表也将不复存在。

Merge 引擎

Merge 引擎是 MyISAM 引擎的一个变种。Merge 表是由多个 MyISAM 表合并而来的虚拟表。如果将 MySQL 用于日志或者数据仓库类应用，该引擎可以发挥作用。但是引入分区功能后，该引擎已经被放弃（参考第 7 章）。

NDB 集群引擎

2003 年，当时的 MySQL AB 公司从索尼爱立信公司收购了 NDB 数据库，然后开发了 NDB 集群存储引擎，作为 SQL 和 NDB 原生协议之间的接口。MySQL 服务器、NDB 集群存储引擎，以及分布式的、share-nothing 的、容灾的、高可用的 NDB 数据库的组合，被称为 MySQL 集群（MySQL Cluster）。本书后续会有章节专门来讨论 MySQL 集群。

1.5.4 第三方存储引擎

MySQL 从 2007 年开始提供了插件式的存储引擎 API，从此涌出了一系列为不同目的而设计的存储引擎。其中有一些已经合并到 MySQL 服务器，但大多数还是第三方产品或者开源项目。下面探讨一些我们认为在它设计的场景中确实很有用的第三方存储引擎。

22

OLTP 类引擎

Percona 的 XtraDB 存储引擎是基于 InnoDB 引擎的一个改进版本，已经包含在 Percona Server 和 MariaDB 中，它的改进点主要集中在性能、可测量性和操作灵活性方面。XtraDB 可以作为 InnoDB 的一个完全的替代产品，甚至可以兼容地读写 InnoDB 的数据文件，并支持 InnoDB 的所有查询。

另外还有一些和 InnoDB 非常类似的 OLTP 类存储引擎，比如都支持 ACID 事务和 MVCC。其中一个就是 PBXT，由 Paul McCullagh 和 Primebase GMBH 开发。它支持引擎级别的复制、外键约束，并且以一种比较复杂的架构对固态存储（SSD）提供了适当的支持，还对较大的值类型如 BLOB 也做了优化。PBXT 是一款社区支持的存储引擎，MariaDB 包含了该引擎。

TokuDB 引擎使用了一种新的叫做分形树（Fractal Trees）的索引数据结构。该结构是缓存无关的，因此即使其大小超过内存性能也不会下降，也就没有内存生命周期和碎片的问题。TokuDB 是一种大数据（Big Data）存储引擎，因为其拥有很高的压缩比，可以在很大的数据量上创建大量索引。在本书写作时，这个引擎还处于早期的生产版本状态，在并发性方面还有很多明显的限制。目前其最适合在需要大量插入数据的分析型数据集的场景中使用，不过这些限制可能在后续版本中解决掉。

RethinkDB 最初是为固态存储（SSD）而设计的，然而随着时间的推移，目前看起来和最初的目标有一定的差距。该引擎比较特别的地方在于采用了一种只能追加的写时复制 B 树（append-only copyon-write B-Tree）作为索引的数据结构。目前还处于早期开发状态，我们还没有测试评估过，也没有听说有实际的应用案例。

在 Sun 收购 MySQL AB 以后，Falcon 存储引擎曾经作为下一代存储引擎被寄予期望，但现在该项目已经被取消很久了。Falcon 的主要设计者 Jim Starkey 创立了一家新公司，主要做可以支持云计算的 NewSQL 数据库产品，叫做 NuoDB（之前叫 NimbusDB）。

面向列的存储引擎

MySQL 默认是面向行的，每一行的数据是一起存储的，服务器的查询也是以行为单位处理的。而在大数据量处理时，面向列的方式可能效率更高。如果不需要整行的数据，

面向列的方式可以传输更少的数据。如果每一列都单独存储，那么压缩的效率也会更高。

Infobright 是最有名的面向列的存储引擎。在非常大的数据量（数十 TB）时，该引擎工作良好。Infobright 是为数据分析和数据仓库应用设计的。数据高度压缩，按照块进行排序，每个块都对应有一组元数据。在处理查询时，访问元数据可决定跳过该块，甚至可能只需要元数据即可满足查询的需求。但该引擎不支持索引，不过在这么大的数据量级，即使有索引也很难发挥作用，而且块结构也是一种准索引（quasi-index）。Infobright 需要对 MySQL 服务器做定制，因为一些地方需要修改以适应面向列存储的需要。如果查询无法在存储层使用面向列的模式执行，则需要在服务器层转换成按行处理，这个过程会很慢。Infobright 有社区版和商业版两个版本。 23

另外一个面向列的存储引擎是 Calpont 公司的 InfiniDB，也有社区版和商业版。InfiniDB 可以在一组机器集群间做分布式查询，但目前还没有生产环境的应用案例。

顺便提一下，在 MySQL 之外，如果有面向列的存储的需求，我们也评估过 LucidDB 和 MonetDB。在我们的 MySQL 性能博客[注5]上有相应的性能测试数据，或许随着时间的推移，这些数据慢慢会过期，但依然可以作为参考。

社区存储引擎

如果要列举社区提供的所有存储引擎，可能会有两位数，甚至三位数。但是负责任地说，其中大部分影响力有限，很多可能都没有听说过，或者只有极少人在使用。在这里列举了一些，也大都没有在生产环境中应用过，慎用，后果自负。

Aria

之前的名字是 Maria，是 MySQL 创建者计划用来替代 MyISAM 的一款引擎。MariaDB 包含了该引擎，之前计划开发的很多特性，有些因为在 MariaDB 服务器层实现，所以引擎层就取消了。在本书写作之际，可以说 Aria 就是解决了崩溃安全恢复问题的 MyISAM，当然也还有一些特性是 MyISAM 不具备的，比如数据的缓存（MyISAM 只能缓存索引）。

Groonga

这是一款全文索引引擎，号称可以提供准确而高效的全文索引。

OQGraph

该引擎由 Open Query 研发，支持图操作（比如查找两点之间的最短路径），用 SQL 很难实现该类操作。

Q4M

该引擎在 MySQL 内部实现了队列操作，而用 SQL 很难在一个语句实现这类队列

注 5： *mysqlperformanceblog.com*。——译者注

操作。

SphinxSE

该引擎为 Sphinx 全文索引搜索服务器提供了 SQL 接口，在附录 F 中将做进一步的详细讨论。

24 *Spider*

该引擎可以将数据切分成不同的分区，比较高效透明地实现了分片（shard），并且可以针对分片执行并行查询（分片可以分布在不同的服务器上）。

VPForMySQL

该引擎支持垂直分区，通过一系列的代理存储引擎实现。垂直分区指的是可以将表分成不同列的组合，并且单独存储。但对查询来说，看到的还是一张表。该引擎和 Spider 的作者是同一人。

1.5.5 选择合适的引擎

这么多存储引擎，我们怎么选择？大部分情况下，InnoDB 都是正确的选择，所以 Oracle 在 MySQL 5.5 版本时终于将 InnoDB 作为默认的存储引擎了。对于如何选择存储引擎，可以简单地归纳为一句话："除非需要用到某些 InnoDB 不具备的特性，并且没有其他办法可以替代，否则都应该优先选择 InnoDB 引擎"。例如，如果要用到全文索引，建议优先考虑 InnoDB 加上 Sphinx 的组合，而不是使用支持全文索引的 MyISAM。当然，如果不需要用到 InnoDB 的特性，同时其他引擎的特性能够更好地满足需求，也可以考虑一下其他存储引擎。举个例子，如果不在乎可扩展能力和并发能力，也不在乎崩溃后的数据丢失问题，却对 InnoDB 的空间占用过多比较敏感，这种场合下选择 MyISAM 就比较合适。

除非万不得已，否则建议不要混合使用多种存储引擎，否则可能带来一系列复杂的问题，以及一些潜在的 bug 和边界问题。存储引擎层和服务器层的交互已经比较复杂，更不用说混合多个存储引擎了。至少，混合存储对一致性备份和服务器参数配置都带来了一些困难。

如果应用需要不同的存储引擎，请先考虑以下几个因素。

事务

如果应用需要事务支持，那么 InnoDB（或者 XtraDB）是目前最稳定并且经过验证的选择。如果不需要事务，并且主要是 SELECT 和 INSERT 操作，那么 MyISAM 是不错的选择。一般日志型的应用比较符合这一特性。

备份

备份的需求也会影响存储引擎的选择。如果可以定期地关闭服务器来执行备份，那

么备份的因素可以忽略。反之，如果需要在线热备份，那么选择 InnoDB 就是基本的要求。

崩溃恢复 25

数据量比较大的时候，系统崩溃后如何快速地恢复是一个需要考虑的问题。相对而言，MyISAM 崩溃后发生损坏的概率比 InnoDB 要高很多，而且恢复速度也要慢。因此，即使不需要事务支持，很多人也选择 InnoDB 引擎，这是一个非常重要的因素。

特有的特性

最后，有些应用可能依赖一些存储引擎所独有的特性或者优化，比如很多应用依赖聚簇索引的优化。另外，MySQL 中也只有 MyISAM 支持地理空间搜索。如果一个存储引擎拥有一些关键的特性，同时却又缺乏一些必要的特性，那么有时候不得不做折中的考虑，或者在架构设计上做一些取舍。某些存储引擎无法直接支持的特性，有时候通过变通也可以满足需求。

你不需要现在就做决定。本书接下来会提供很多关于各种存储引擎优缺点的详细描述，也会讨论一些架构设计的技巧。一般来说，可能有很多选项你还没有意识到，等阅读完本书回头再来看这个问题可能更有帮助些。如果无法确定，那么就使用 InnoDB，这个默认选项是安全的，尤其是搞不清楚具体需要什么的时候。

如果不了解具体的应用，上面提到的这些概念都是比较抽象的。所以接下来会讨论一些常见的应用场景，在这些场景中会涉及很多的表，以及这些表如何选用合适的存储引擎，下一节将进行一些总结。

日志型应用

假设你需要实时地记录一台中心电话交换机的每一通电话的日志到 MySQL 中，或者通过 Apache 的 *mod_log_sql* 模块将网站的所有访问信息直接记录到表中。这一类应用的插入速度有很高的要求，数据库不能成为瓶颈。MyISAM 或者 Archive 存储引擎对这类应用比较合适，因为它们开销低，而且插入速度非常快。

如果需要对记录的日志做分析报表，则事情就会变得有趣了。生成报表的 SQL 很有可能会导致插入效率明显降低，这时候该怎么办？

一种解决方法，是利用 MySQL 内置的复制方案将数据复制一份到备库，然后在备库上执行比较消耗时间和 CPU 的查询。这样主库只用于高效的插入工作，而备库上执行的查询也无须担心影响到日志的插入性能。当然也可以在系统负载较低的时候执行报表查询 26 操作，但应用在不断变化，如果依赖这个策略可能以后会导致问题。

另外一种方法，在日志记录表的名字中包含年和月的信息，比如 `web_logs_2012_01` 或者

web_logs_2012_jan。这样可以在已经没有插入操作的历史表上做频繁的查询操作，而不会干扰到最新的当前表上的插入操作。

只读或者大部分情况下只读的表

有些表的数据用于编制类目或者分列清单（如工作岗位、竞拍、不动产等），这种应用场景是典型的读多写少的业务。如果不介意 MyISAM 的崩溃恢复问题，选用 MyISAM 引擎是合适的。不过不要低估崩溃恢复问题的重要性，有些存储引擎不会保证将数据安全地写入到磁盘中，而许多用户实际上并不清楚这样有多大的风险（MyISAM 只将数据写到内存中，然后等待操作系统定期将数据刷出到磁盘上）。

> 一个值得推荐的方式，是在性能测试环境模拟真实的环境，运行应用，然后拔下电源模拟崩溃测试。对崩溃恢复的第一手测试经验是无价之宝，可以避免真的碰到崩溃时手足无措。

不要轻易相信"MyISAM 比 InnoDB 快"之类的经验之谈，这个结论往往不是绝对的。在很多我们已知的场景中，InnoDB 的速度都可以让 MyISAM 望尘莫及，尤其是使用到聚簇索引，或者需要访问的数据都可以放入内存的应用。在本书后续章节，读者可以了解更多影响存储引擎性能的因素（如数据大小、I/O 请求量、主键还是二级索引等）以及这些因素对应用的影响。

当设计上述类型的应用时，建议采用 InnoDB。MyISAM 引擎在一开始可能没有任何问题，但随着应用压力的上升，则可能迅速恶化。各种锁争用、崩溃后的数据丢失等问题都会随之而来。

订单处理

如果涉及订单处理，那么支持事务就是必要选项。半完成的订单是无法用来吸引用户的。另外一个重要的考虑点是存储引擎对外键的支持情况。InnoDB 是订单处理类应用的最佳选择。

27 电子公告牌和主题讨论论坛

对于 MySQL 用户，主题讨论区是个很有意思的话题。当前有成百上千的基于 PHP 或者 Perl 的免费系统可以支持主题讨论。其中大部分的数据库操作效率都不高，因为它们大多倾向于在一次请求中执行尽可能多的查询语句。另外还有部分系统设计为不采用数据库，当然也就无法利用到数据库提供的一些方便的特性。主题讨论区一般都有更新计数器，并且会为各个主题计算访问统计信息。多数应用只设计了几张表来保存所有的数据，

所以核心表的读写压力可能非常大。为保证这些核心表的数据一致性，锁成为资源争用的主要因素。

尽管有这些设计缺陷，但大多数应用在中低负载时可以工作得很好。如果 Web 站点的规模迅速扩展，流量随之猛增，则数据库访问可能变得非常慢。此时一个典型的解决方案是更改为支持更高读写的存储引擎，但有时用户会发现这么做反而导致系统变得更慢了。

用户可能没有意识到这是由于某些特殊查询的缘故，典型的如：

```
mysql> SELECT COUNT(*) FROM table;
```

问题就在于，不是所有的存储引擎运行上述查询都非常快：对于 MyISAM 确实会很快，但其他的可能都不行。每种存储引擎都能找出类似的对自己有利的例子。下一章将帮助用户分析这些状况，演示如何发现和解决存在的这类问题。

CD-ROM 应用

如果要发布一个基于 CD-ROM 或者 DVD-ROM 并且使用 MySQL 数据文件的应用，可以考虑使用 MyISAM 表或者 MyISAM 压缩表，这样表之间可以隔离并且可以在不同介质上相互拷贝。MyISAM 压缩表比未压缩的表要节约很多空间，但压缩表是只读的。在某些应用中这可能是个大问题。但如果数据放到只读介质的场景下，压缩表的只读特性就不是问题，就没有理由不采用压缩表了。

大数据量

什么样的数据量算大？我们创建或者管理的很多 InnoDB 数据库的数据量在 3 ~ 5TB 之间，或者更大，这是单台机器上的量，不是一个分片（shard）的量。这些系统运行得还不错，要做到这一点需要合理地选择硬件，做好物理设计，并为服务器的 I/O 瓶颈做好规划。在这样的数据量下，如果采用 MyISAM，崩溃后的恢复就是一个噩梦。

如果数据量继续增长到 10TB 以上的级别，可能就需要建立数据仓库。Infobright 是 MySQL 数据仓库最成功的解决方案。也有一些大数据库不适合 Infobright，却可能适合 28
TokuDB。

1.5.6 转换表的引擎

有很多种方法可以将表的存储引擎转换成另外一种引擎。每种方法都有其优点和缺点。在接下来的章节中，我们将讲述其中的三种方法。

ALTER TABLE

将表从一个引擎修改为另一个引擎最简单的办法是使用 ALTER TABLE 语句。下面的语句将 mytable 的引擎修改为 InnoDB：

```
mysql> ALTER TABLE mytable ENGINE = InnoDB;
```

上述语法可以适用任何存储引擎。但有一个问题：需要执行很长时间。MySQL 会按行将数据从原表复制到一张新的表中，在复制期间可能会消耗系统所有的 I/O 能力，同时原表上会加上读锁。所以，在繁忙的表上执行此操作要特别小心。一个替代方案是采用接下来将讨论的导出与导入的方法，手工进行表的复制。

如果转换表的存储引擎，将会失去和原引擎相关的所有特性。例如，如果将一张 InnoDB 表转换为 MyISAM，然后再转换回 InnoDB，原 InnoDB 表上所有的外键将丢失。

导出与导入

为了更好地控制转换的过程，可以使用 *mysqldump* 工具将数据导出到文件，然后修改文件中 CREATE TABLE 语句的存储引擎选项，注意同时修改表名，因为同一个数据库中不能存在相同的表名，即使它们使用的是不同的存储引擎。同时要注意 *mysqldump* 默认会自动在 CREATE TABLE 语句前加上 DROP TABLE 语句，不注意这一点可能会导致数据丢失。

创建与查询（CREATE 和 SELECT）

第三种转换的技术综合了第一种方法的高效和第二种方法的安全。不需要导出整个表的数据，而是先创建一个新的存储引擎的表，然后利用 INSERT…SELECT 语法来导数据：

```
mysql> CREATE TABLE innodb_table LIKE myisam_table;
mysql> ALTER TABLE innodb_table ENGINE=InnoDB;
mysql> INSERT INTO innodb_table SELECT * FROM myisam_table;
```

29 数据量不大的话，这样做工作得很好。如果数据量很大，则可以考虑做分批处理，针对每一段数据执行事务提交操作，以避免大事务产生过多的 undo。假设有主键字段 id，重复运行以下语句（最小值 x 和最大值 y 进行相应的替换）将数据导入到新表：

```
mysql> START TRANSACTION;
mysql> INSERT INTO innodb_table SELECT * FROM myisam_table
    -> WHERE id BETWEEN x AND y;
mysql> COMMIT;
```

这样操作完成以后，新表是原表的一个全量复制，原表还在，如果需要可以删除原表。如果有必要，可以在执行的过程中对原表加锁，以确保新表和原表的数据一致。

Percona Toolkit 提供了一个 *pt-online-schema-change* 的工具（基于 Facebook 的在线

schema 变更技术），可以比较简单、方便地执行上述过程，避免手工操作可能导致的失误和烦琐。

1.6 MySQL 时间线（Timeline）

在选择 MySQL 版本的时候，了解一下版本的变迁历史是有帮助的。对于怀旧者也可以享受一下过去的好日子里是怎么使用 MySQL 的。

版本 3.23（2001）

一般认为这个版本的发布是 MySQL 真正"诞生"的时刻，其开始获得广泛使用。在这个版本，MySQL 依然只是一个在平面文件（Flat File）上实现了 SQL 查询的系统。但一个重要的改进是引入 MyISAM 代替了老旧而且有诸多限制的 ISAM 引擎。InnoDB 引擎也已经可以使用，但没有包含在默认的二进制发行版中，因为它太新了。所以如果要使用 InnoDB，必须手工编译。版本 3.23 还引入了全文索引和复制。复制是 MySQL 成为互联网应用的数据库系统的关键特性（killer feature）。

版本 4.0（2003）

支持新的语法，比如 `UNION` 和多表 `DELETE` 语法。重写了复制，在备库使用了两个线程来实现复制，避免了之前一个线程做所有复制工作的模式下任务切换导致的问题。InnoDB 成为标准配备，包括了全部的特性：行级锁、外键等。版本 4.0 中还引入了查询缓存（自那以后这部分改动不大），同时还支持通过 SSL 进行连接。

版本 4.1（2005）

引入了更多新的语法，比如子查询和 `INSERT ON DUPLICATE KEY UPDATE`。开始支持 UTF-8 字符集。支持新的二进制协议和 prepared 语句。

版本 5.0（2006） 30

这个版本出现了一些"企业级"特性：视图、触发器、存储过程和存储函数。老的 ISAM 引擎的代码被彻底移除，同时引入了新的 Federated 等引擎。

版本 5.1（2008）

这是 Sun 收购 MySQL AB 以后发布的首个版本，研发时间长达五年。版本 5.1 引入了分区、基于行的复制，以及 plugin API（包括可插拔存储引擎的 API）。移除了 BerkeyDB 引擎，这是 MySQL 最早的事务存储引擎。其他如 Federated 引擎也将被放弃。同时 Oracle 收购的 InnoDB Oy[注6] 发布了 InnoDB plugin。

版本 5.5（2010）

这是 Oracle 收购 Sun 以后发布的首个版本。版本 5.5 的主要改善集中在性能、扩展性、复制、分区、对微软 Windows 系统的支持，以及一些其他方面。InnoDB 成为默认的存储引擎。更多的一些遗留特性和不建议使用的特性被移除。增加了

注 6： Oracle 也已经收购了 BerkeyDB。

PERFORMANCE_SCHEMA 库，包含了一些可测量的性能指标的增强。增加了复制、认证和审计 API。半同步复制（semisynchronous replication）插件进入实用阶段。Oracle 还在 2011 年发布了商用的认证插件和线程池（thread pooling）。InnoDB 在架构方面也做了较大的改进，比如多个子缓冲池（buffer pool）。

版本 5.6（还未发布）

版本 5.6 将包含一些重大更新。比如多年来首次对查询优化器进行大规模的改进，更多的插件 API（比如全文索引），复制的改进，以及 PERFORMANCE_SCHEMA 库增加了更多的性能指标。InnoDB 团队也做了大量的改进工作，这些改进在已经发布的里程碑版本和实验室版本中都已经包括。MySQL 5.5 主要着重在基础部分的改进和加强，引入了部分新特性。而 MySQL 5.6 则在 MySQL 5.5 的基础上提升服务器的开发和性能。

版本 6.0（已经取消）

版本 6.0 的概念有些模糊。最早在版本 5.1 还在开发的时候就宣布要开发版本 6.0。传说中宣布要开发的 6.0 拥有大量的新特性，包括在线备份、服务器层面对所有存储引擎的外键支持，以及子查询的改进和线程池。后来该版本号被取消，Sun 将其
31 改为版本 5.4 继续开发，最后发布时变成版本 5.5。版本 6.0 中很多特性的代码陆续出现在版本 5.5 和 5.6 中。

简单总结一下 MySQL 的发展史：早期的 MySQL 是一种破坏性创新[注7]，有诸多限制，并且很多功能只能说是二流的。但是它的特性支持和较低的使用成本，使得其成为快速增长的互联网时代的杀手级应用。在 5.x 版本的早期，MySQL 引入了视图和存储过程等特性，期望成为“企业级”数据库，但并不算成功，成长并非一帆风顺。从事后分析来看，MySQL 5.0 充满了 bug，直到 5.0.50 以后的版本才算稳定。这种情况在 MySQL 5.1 也依然没有太多改善。版本 5.0 和 5.1 的发布都延期了许多时日，而且 Sun 和 Oracle 的两次收购也使得社区人士有所担心。但我们认为事情还在按部就班地发展，MySQL 5.5 可以说是 MySQL 历史上质量最高的版本。Oracle 收购以后帮助 MySQL 更好地往企业级应用的方向发展，MySQL 5.6 也承诺在功能和性能方面将有显著提升。

提到性能，我们可以比较一下在不同时代 MySQL 的性能测试的数据。在目前的生产环境中 4.0 及更老的版本已经很少见了，所以这里不打算测试 4.1 之前的版本。另外，如此多的版本如果要做完全等同的测试是比较困难的，具体原因将在后面的章节讨论。我们尝试设计了多个测试方案来尽量保证在不同版本中的基准一致，并为此做了很多努力。表 1-2 显示了在服务器层面不同并发下的每秒事务数的测试结果。

注 7：“破坏性创新”一词出自 Clayton M. Christensen 的 *The Innovator's Dilemma*（Harper）。

表1-2：多个不同MySQL版本的只读测试

线程数	MySQL 4.1	MySQL 5.0	MySQL 5.1	MySQL 5.1 with InnoDB plugin	MySQL 5.5	MySQL 5.6[a]
1	686	640	596	594	531	526
2	1307	1221	1140	1139	1077	1019
4	2275	2168	2032	2043	1938	1831
8	3879	3746	3606	3681	3523	3320
16	4374	4527	4393	6131	5881	5573
32	4591	4864	4698	7762	7549	7139
64	4688	5078	4910	7536	7269	6994

注 a：在测试的时候，版本 5.6 还没有 GA（正式发布）。

很容易将表 1-2 的数据以图的方式展示出来，如图 1-2 所示。

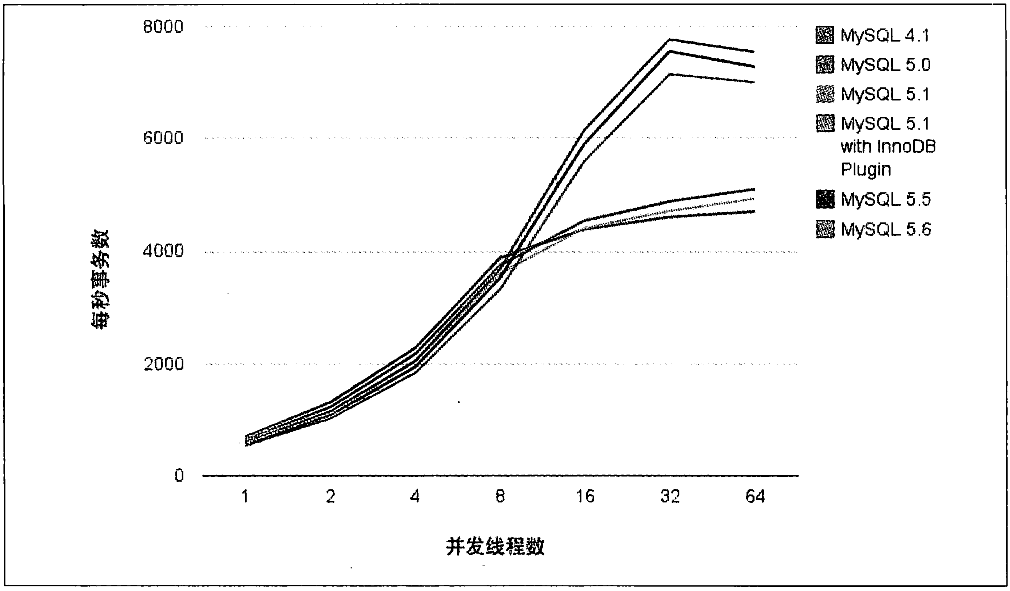


图1-2：MySQL不同版本的只读基准测试

在解释结果之前，需要先介绍一下测试环境。测试的机器是 Cisco UCS C250，两颗 6 核 CPU，每个核支持两个线程，内存为 384GB，测试的数据集是 2.5GB，所以 MySQL 的 buffer pool 设置为 4GB。采用 SysBench 的 read-only 只读测试进行压测，并采用 InnoDB 存储引擎，所有的数据都可以放入内存，因此是 CPU 密集型（CPU-bound）的测试。每次测试持续 60 分钟，每 10 秒获取一次吞吐量的结果，前面 900 秒用于预热数据，以避免预热时的 I/O 影响测试结果。 32

现在来看看结果，有两个很明显的趋势。第一个趋势，采用了 InnoDB plugin 的版本，在高并发的时候性能明显更好，可以说 InnoDB plugin 的扩展性更好。这是可以预期的结果，旧的版本在高并发时确实存在问题。第二个趋势，新的版本在单线程的时候性能比旧版本更差。一开始可能无法理解为什么会这样，仔细想想就能明白，这是一个非常简单的只读测试。新版本的 SQL 语法更复杂，针对复杂查询增加了很多特性和改进，这对于简单查询可能带来了更多的开销。旧版本的代码简单，对于简单的查询反而会更有利。

原计划做一个更复杂的不同并发条件下的读写混合场景的测试（类似 TPC-C），但要在不同版本间做到可比较基本是不可能的。一般来说，新版本在复杂场景时性能有更多的优化，尤其是高并发和大数据集的情况下。

33 那么该如何选择版本呢？这更多地取决于业务需求而不是技术需求。理想情况下当然是版本越新越好，当然也可以选择等到第一个 bug 修复版本以后再采用新的大版本。如果应用还没有上线，也可以采用即将发布的新版本，以尽可能地延迟应用上线后的升级操作。

1.7 MySQL 的开发模式

MySQL 的开发过程和发布模型在不同的阶段有很大的变化，但目前已经基本稳定下来。在 Oracle 定期发布的新里程碑开发版本中，会包含即将在下一个 GA[注8] 版本发布的新特性。这样做是为了测试和获得反馈，请不要在生产环境使用此版本，虽然 Oracle 宣称每个里程碑版本的质量都是可靠的，并随时可以正式发布（到目前为止也没有任何理由去推翻这个说法）。Oracle 也会定期发布实验室预览版，主要包含一些特定的需要评估的特性，这些特性并不保证会在下一个正式版本中包括进去。最终，Oracle 会将稳定的特性打包发布一个新的 GA 版本。

MySQL 依然遵循 GPL 开源协议，全部的源代码（除了一些商业版本的插件）都会开放给社区。Oracle 似乎也理解，为社区和付费用户提供不同的版本并非明智之举。MySQL AB 曾经尝试过不同版本的策略，结果导致付费用户变成了“睁眼瞎”，无法从社区的测试和反馈中获得好处。不同版本的策略并不受企业用户的欢迎，所以后来被 Sun 废除了。

现在 Oracle 为付费用户单独提供了一些服务器插件，而 MySQL 本身还是遵循开源模式。尽管对于私有的服务器插件的发布有一些抱怨，但这只是少数的声音，并且慢慢地在平息。大多数 MySQL 用户对此并不在意，有需求的用户也能够接受商业授权的付费插件。

无论如何，不开源的扩展也只是扩展而已，并不会将 MySQL 变成受限制的非开源模式。

注 8： GA (Generally Available) 的意思是通常可用的版本，对于最挑剔的老板来说，这种版本也意味着达到了满足生产环境中使用的质量标准。

没有这些扩展，MySQL 也是功能完整的数据库。坦白地说，我们也很欣赏 Oracle 将更多的特性做成插件的开发模式。如果将特性直接包含在服务器中而不是 API 的方式，那就更加没有选择了：用户只能接受这种实现，而失去了选择更适合业务的实现的机会。例如，如果 Oracle 将 InnoDB 的全文索引功能以 API 的方式实现，那么就可能以同样的 API 实现 Sphinx 或者 Lucene 的插件，这可能对一些用户更有用。服务器内部的 API 设计也很干净，这对于提升代码质量非常有帮助，谁不想要这个呢？ 34

1.8 总结

MySQL 拥有分层的架构。上层是服务器层的服务和查询执行引擎，下层则是存储引擎。虽然有很多不同作用的插件 API，但存储引擎 API 还是最重要的。如果能理解 MySQL 在存储引擎和服务层之间处理查询时如何通过 API 来回交互，就能抓住 MySQL 的核心基础架构的精髓。

MySQL 最初基于 ISAM 构建（后来被 MyISAM 取代），其后陆续添加了更多的存储引擎和事务支持。MySQL 有一些怪异的行为是由于历史遗留导致的。例如，在执行 ALTER TABLE 时，MySQL 提交事务的方式是由于存储引擎的架构直接导致的，并且数据字典也保存在 *.frm* 文件中（这并不是说 InnoDB 会导致 ALTER 变成非事务型的。对于 InnoDB 来说，所有的操作都是事务）。

当然，存储引擎 API 的架构也有一些缺点。有时候选择多并非好事，而在 MySQL 5.0 和 MySQL 5.1 中有太多的存储引擎可以选择。InnoDB 对于 95% 以上的用户来说都是最佳选择，所以其他的存储引擎可能只是让事情变得复杂难搞，当然也不可否认某些情况下某些存储引擎能更好地满足需求。

Oracle 一开始收购了 InnoDB，之后又收购了 MySQL，在同一个屋檐下对于两者都是有利的。InnoDB 和 MySQL 服务器之间可以更快地协同发展。MySQL 依然基于 GPL 协议开放全部源代码，社区和客户都可以获得坚固而稳定的数据库，MySQL 正在变得越来越可扩展和有用。

第 2 章 35

MySQL基准测试

基准测试（benchmark）是 MySQL 新手和专家都需要掌握的一项基本技能。简单地说，基准测试是针对系统设计的一种压力测试。通常的目标是为了掌握系统的行为。但也有其他原因，如重现某个系统状态，或者是做新硬件的可靠性测试。本章将讨论 MySQL 和基于 MySQL 的应用的基准测试的重要性、策略和工具。我们将特别讨论一下 sysbench，这是一款非常优秀的 MySQL 基准测试工具。

2.1 为什么需要基准测试

为什么基准测试很重要？因为基准测试是唯一方便有效的、可以学习系统在给定的工作负载下会发生什么的方法。基准测试可以观察系统在不同压力下的行为，评估系统的容量，掌握哪些是重要的变化，或者观察系统如何处理不同的数据。基准测试可以在系统实际负载之外创造一些虚构场景进行测试。基准测试可以完成以下工作，或者更多：

- 验证基于系统的一些假设，确认这些假设是否符合实际情况。
- 重现系统中的某些异常行为，以解决这些异常。
- 测试系统当前的运行情况。如果不清楚系统当前的性能，就无法确认某些优化的效果如何。也可以利用历史的基准测试结果来分析诊断一些无法预测的问题。
- 模拟比当前系统更高的负载，以找出系统随着压力增加而可能遇到的扩展性瓶颈。
- 规划未来的业务增长。基准测试可以评估在项目未来的负载下，需要什么样的硬件，需要多大容量的网络，以及其他相关资源。这有助于降低系统升级和重大变更的风险。
- 测试应用适应可变环境的能力。例如，通过基准测试，可以发现系统在随机的并发 36
  峰值下的性能表现，或者是不同配置的服务器之间的性能表现。基准测试也可以测试系统对不同数据分布的处理能力。

- 测试不同的硬件、软件和操作系统配置。比如 RAID 5 还是 RAID 10 更适合当前的系统？如果系统从 ATA 硬盘升级到 SAN 存储，对于随机写性能有什么帮助？ Linux 2.4 系列的内核会比 2.6 系列的可扩展性更好吗？升级 MySQL 的版本能改善性能吗？为当前的数据采用不同的存储引擎会有什么效果？所有这类问题都可以通过专门的基准测试来获得答案。
- 证明新采购的设备是否配置正确。笔者曾经无数次地通过基准测试来对新系统进行压测，发现了很多错误的配置，以及硬件组件的失效等问题。因此在新系统正式上线到生产环境之前进行基准测试是一个好习惯，永远不要相信主机提供商或者硬件供应商的所谓系统已经安装好，并且能运行多快的说法。如果可能，执行实际的基准测试永远是一个好主意。

基准测试还可以用于其他目的，比如为应用创建单元测试套件。但本章我们只关注与性能有关的基准测试。

基准测试的一个主要问题在于其不是真实压力的测试。基准测试施加给系统的压力相对真实压力来说，通常比较简单。真实压力是不可预期而且变化多端的，有时候情况会过于复杂而难以解释。所以使用真实压力测试，可能难以从结果中分析出确切的结论。

基准测试的压力和真实压力在哪些方面不同？有很多因素会影响基准测试，比如数据量、数据和查询的分布，但最重要的一点还是基准测试通常要求尽可能快地执行完成，所以经常给系统造成过大的压力。在很多案例中，我们都会调整给测试工具的最大压力，以在系统可以容忍的压力阈值内尽可能快地执行测试，这对于确定系统的最大容量非常有帮助。然而大部分压力测试工具不支持对压力进行复杂的控制。务必要记住，测试工具自身的局限也会影响到结果的有效性。

使用基准测试进行容量规划也要掌握技巧，不能只根据测试结果做简单的推断。例如，假设想知道使用新数据库服务器后，系统能够支撑多大的业务增长。首先对原系统进行基准测试，然后对新系统做测试，结果发现新系统可以支持原系统 40 倍的 TPS（每秒事务数），这时候就不能简单地推断说新系统一定可以支持 40 倍的业务增长。这是因为在业务增长的同时，系统的流量、用户、数据以及不同数据之间的交互都在增长，它们不可能都有 40 倍的支撑能力，尤其是相互之间的关系。而且当业务增长到 40 倍时，应
37 用本身的设计也可能已经随之改变。可能有更多的新特性会上线，其中某些特性可能对数据库造成的压力远大于原有功能。而这些压力、数据、关系和特性的变化都很难模拟，所以它们对系统的影响也很难评估。

结论就是，我们只能进行大概的测试，来确定系统大致的余量有多少。当然也可以做一些真实压力测试（和基准测试有区别），但在构造数据集和压力的时候要特别小心，而

且这样就不再是基准测试了。基准测试要尽量简单直接，结果之间容易相互比较，成本低且易于执行。尽管有诸多限制，基准测试还是非常有用的（只要搞清楚测试的原理，并且了解如何分析结果所代表的意义）。

2.2 基准测试的策略

基准测试有两种主要的策略：一是针对整个系统的整体测试，另外是单独测试 MySQL。这两种策略也被称为集成式（full-stack）以及单组件式（single-component）基准测试。针对整个系统做集成式测试，而不是单独测试 MySQL 的原因主要有以下几点：

- 测试整个应用系统，包括 Web 服务器、应用代码、网络和数据库是非常有用的，因为用户关注的并不仅仅是 MySQL 本身的性能，而是应用整体的性能。
- MySQL 并非总是应用的瓶颈，通过整体的测试可以揭示这一点。
- 只有对应用做整体测试，才能发现各部分之间的缓存带来的影响。
- 整体应用的集成式测试更能揭示应用的真实表现，而单独组件的测试很难做到这一点。

另外一方面，应用的整体基准测试很难建立，甚至很难正确设置。如果基准测试的设计有问题，那么结果就无法反映真实的情况，从而基于此做的决策也就可能是错误的。

不过，有时候不需要了解整个应用的情况，而只需要关注 MySQL 的性能，至少在项目初期可以这样做。基于以下情况，可以选择只测试 MySQL：

- 需要比较不同的 schema 或查询的性能。
- 针对应用中某个具体问题的测试。
- 为了避免漫长的基准测试，可以通过一个短期的基准测试，做快速的“周期循环”，来检测出某些调整后的效果。

另外，如果能够在真实的数据集上执行重复的查询，那么针对 MySQL 的基准测试也是有用的，但是数据本身和数据集的大小都应该是真实的。如果可能，可以采用生产环境的数据快照。

不幸的是，设置一个基于真实数据的基准测试复杂而且耗时。如果能得到一份生产数据 38
集的拷贝，当然很幸运，但这通常不太可能。比如要测试的是一个刚开发的新应用，它只有很少的用户和数据。如果想测试该应用在规模扩张到很大以后的性能表现，就只能通过模拟大量的数据和压力来进行。

2.2.1 测试何种指标

在开始执行甚至是在设计基准测试之前，需要先明确测试的目标。测试目标决定了选择什么样的测试工具和技术，以获得精确而有意义的测试结果。可以将测试目标细化为一系列的问题，比如，“这种 CPU 是否比另外一种要快？”，或“新索引是否比当前索引性能更好？”

有时候需要用不同的方法测试不同的指标。比如，针对延迟（latency）和吞吐量（throughput）就需要采用不同的测试方法。

请考虑以下指标，看看如何满足测试的需求。

吞吐量

吞吐量指的是单位时间内的事务处理数。这一直是经典的数据库应用测试指标。一些标准的基准测试被广泛地引用，如 TPC-C（参考 *http://www.tpc.org*），而且很多数据库厂商都努力争取在这些测试中取得好成绩。这类基准测试主要针对在线事务处理（OLTP）的吞吐量，非常适用于多用户的交互式应用。常用的测试单位是每秒事务数（TPS），有些也采用每分钟事务数（TPM）。

响应时间或者延迟

这个指标用于测试任务所需的整体时间。根据具体的应用，测试的时间单位可能是微秒、毫秒、秒或者分钟。根据不同的时间单位可以计算出平均响应时间、最小响应时间、最大响应时间和所占百分比。最大响应时间通常意义不大，因为测试时间越长，最大响应时间也可能越大。而且其结果通常不可重复，每次测试都可能得到不同的最大响应时间。因此，通常可以使用百分比响应时间（percentile response time）来替代最大响应时间。例如，如果 95% 的响应时间都是 5 毫秒，则表示任务在 95% 的时间段内都可以在 5 毫秒之内完成。

使用图表有助于理解测试结果。可以将测试结果绘制成折线图（比如平均值折线或者 95% 百分比折线）或者散点图，直观地表现数据结果集的分布情况。通过这些图可以发现长时间测试的趋势。本章后面将更详细地讨论这一点。

39 *并发性*

并发性是一个非常重要又经常被误解和误用的指标。例如，它经常被表示成多少用户在同一时间浏览一个 Web 站点，经常使用的指标是有多少个会话[注1]。然而，HTTP 协议是无状态的，大多数用户只是简单地读取浏览器上显示的信息，这并不等同于 Web 服务器的并发性。而且，Web 服务器的并发性也不等同于数据库的并发性，而仅仅只表示会话存储机制可以处理多少数据的能力。Web 服务器的并发性更准确的度量指标，应该是在任意时间有多少同时发生的并发请求。

注 1： 特别是一些论坛软件，已经让很多管理员错误地相信同时有成千上万的用户正在同时访问网站。

在应用的不同环节都可以测量相应的并发性。Web 服务器的高并发，一般也会导致数据库的高并发，但服务器采用的语言和工具集对此都会有影响。注意不要将创建数据库连接和并发性搞混淆。一个设计良好的应用，同时可以打开成百上千个 MySQL 数据库服务器连接，但可能同时只有少数连接在执行查询。所以说，一个 Web 站点“同时有 50 000 个用户”访问，却可能只有 10 ~ 15 个并发请求到 MySQL 数据库。

换句话说，并发性基准测试需要关注的是正在工作中的并发操作，或者是同时工作中的线程数或者连接数。当并发性增加时，需要测量吞吐量是否下降，响应时间是否变长，如果是这样，应用可能就无法处理峰值压力。

并发性的测量完全不同于响应时间和吞吐量。它不像是一个结果，而更像是设置基准测试的一种属性。并发性测试通常不是为了测试应用能达到的并发度，而是为了测试应用在不同并发下的性能。当然，数据库的并发性还是需要测量的。可以通过 *sysbench* 指定 32、64 或者 128 个线程的测试，然后在测试期间记录 MySQL 数据库的 `Threads_running` 状态值。在第 11 章将讨论这个指标对容量规划的影响。

可扩展性

在系统的业务压力可能发生变化的情况下，测试可扩展性就非常必要了。第 11 章将更进一步讨论可扩展性的话题。简单地说，可扩展性指的是，给系统增加一倍的工作，在理想情况下就能获得两倍的结果（即吞吐量增加一倍）。或者说，给系统增加一倍的资源（比如两倍的 CPU 数），就可以获得两倍的吞吐量。当然，同时性能（响应时间）也必须在可以接受的范围内。大多数系统是无法做到如此理想的线性扩展的。 40
随着压力的变化，吞吐量和性能都可能越来越差。

可扩展性指标对于容量规范非常有用，它可以提供其他测试无法提供的信息，来帮助发现应用的瓶颈。比如，如果系统是基于单个用户的响应时间测试（这是一个很糟糕的测试策略）设计的，虽然测试的结果很好，但当并发度增加时，系统的性能有可能变得非常糟糕。而一个基于不断增加用户连接的情况下的响应时间测试则可以发现这个问题。

一些任务，比如从细粒度数据创建汇总表的批量工作，需要的是周期性的快速响应时间。当然也可以测试这些任务纯粹的响应时间，但要注意考虑这些任务之间的相互影响。批量工作可能导致相互之间有影响的查询性能变差，反之亦然。

归根结底，应该测试那些对用户来说最重要的指标。因此应该尽可能地去收集一些需求，比如，什么样的响应时间是可以接受的，期待多少的并发性，等等。然后基于这些需求来设计基准测试，避免目光短浅地只关注部分指标，而忽略其他指标。

2.3 基准测试方法

在了解基本概念之后，现在可以来具体讨论一下如何设计和执行基准测试。但在讨论如何设计好的基准测试之前，先来看一下如何避免一些常见的错误，这些错误可能导致测试结果无用或者不精确：

- 使用真实数据的子集而不是全集。例如应用需要处理几百 GB 的数据，但测试只有 1GB 数据；或者只使用当前数据进行测试，却希望模拟未来业务大幅度增长后的情况。
- 使用错误的数据分布。例如使用均匀分布的数据测试，而系统的真实数据有很多热点区域（随机生成的测试数据通常无法模拟真实的数据分布）。
- 使用不真实的分布参数，例如假定所有用户的个人信息（profile）都会被平均地读取[注2]。
- 在多用户场景中，只做单用户的测试。
- 在单服务器上测试分布式应用。
- 与真实用户行为不匹配。例如 Web 页面中的"思考时间"。真实用户在请求到一个页面后会阅读一段时间，而不是不停顿地一个接一个点击相关链接。
- 41 反复执行同一个查询。真实的查询是不尽相同的，这可能会导致缓存命中率降低。而反复执行同一个查询在某种程度上，会全部或者部分缓存结果。
- 没有检查错误。如果测试的结果无法得到合理的解释，比如一个本应该很慢的查询突然变快了，就应该检查是否有错误产生。否则可能只是测试了 MySQL 检测语法错误的速度了。基准测试完成后，一定要检查一下错误日志，这应当是基本的要求。
- 忽略了系统预热（warm up）的过程。例如系统重启后马上进行测试。有时候需要了解系统重启后需要多长时间才能达到正常的性能容量，要特别留意预热的时长。反过来说，如果要想分析正常的性能，需要注意，若基准测试在重启以后马上启动，则缓存是冷的、还没有数据，这时即使测试的压力相同，得到的结果也和缓存已经装满数据时是不同的。
- 使用默认的服务器配置。第 3 章将详细地讨论服务器的优化配置。
- 测试时间太短。基准测试需要持续一定的时间。后面会继续讨论这个话题。

只有避免了上述错误，才能走上改进测试质量的漫漫长路。

如果其他条件相同，就应努力使测试过程尽可能地接近真实应用的情况。当然，有时候和真实情况稍有些出入问题也不大。例如，实际应用服务器和数据库服务器分别部署在不同的机器。如果采用和实际部署完全相同的配置当然更真实，但也会引入更多的变化因素，比如加入了网络的负载和速度等。而在单一节点上运行测试相对要容易，在某些

注 2： Justin Bieber，我们爱你。这只是开个玩笑。

情况下结果也可以接受，那么就可以在单一节点上进行测试。当然，这样的选择需要根据实际情况来分析是否合适。

2.3.1 设计和规划基准测试

规划基准测试的第一步是提出问题并明确目标。然后决定是采用标准的基准测试，还是设计专用的测试。

如果采用标准的基准测试，应该确认选择了合适的测试方案。例如，不要使用 TPC-H 测试电子商务系统。在 TPC 的定义中，"TPC-H 是即席查询和决策支持型应用的基准测试"，因此不适合用来测试 OLTP 系统。

设计专用的基准测试是很复杂的，往往需要一个迭代的过程。首先需要获得生产数据集的快照，并且该快照很容易还原，以便进行后续的测试。

然后，针对数据运行查询。可以建立一个单元测试集作为初步的测试，并运行多遍。但是这和真实的数据库环境还是有差别的。更好的办法是选择一个有代表性的时间段，比 42
如高峰期的一个小时，或者一整天，记录生产系统上的所有查询。如果时间段选得比较小，则可以选择多个时间段。这样有助于覆盖整个系统的活动状态，例如每周报表的查询、或者非峰值时间运行的批处理作业[注3]。

可以在不同级别记录查询。例如，如果是集成式（full-stack）基准测试，可以记录 Web 服务器上的 HTTP 请求，也可以打开 MySQL 的查询日志（Query Log）。倘若要重演这些查询，就要确保创建多线程来并行执行，而不是单个线程线性地执行。对日志中的每个连接都应该创建独立的线程，而不是将所有的查询随机地分配到一些线程中。查询日志中记录了每个查询是在哪个连接中执行的。

即使不需要创建专用的基准测试，详细地写下测试规划也是必需的。测试可能要多次反复运行，因此需要精确地重现测试过程。而且也应该考虑到未来，执行下一轮测试时可能已经不是同一个人了。即使还是同一个人，也有可能不会确切地记得初次运行时的情况。测试规划应该记录测试数据、系统配置的步骤、如何测量和分析结果，以及预热的方案等。

应该建立将参数和结果文档化的规范，每一轮测试都必须进行详细记录。文档规范可以很简单，比如采用电子表格（spreadsheet）或者记事本形式，也可以是复杂的自定义的数据库。需要记住的是，经常要写一些脚本来分析测试结果，因此如果能够不用打开电子表格或者文本文件等额外操作，当然是更好的。

注 3： 当然，做这么多的前提是希望获得完美的基准测试结果，实际情况通常不会很顺利。

2.3.2 基准测试应该运行多长时间

基准测试应该运行足够长的时间，这一点很重要。如果需要测试系统在稳定状态时的性能，那么当然需要在稳定状态下测试并观察。而如果系统有大量的数据和内存，要达到稳定状态可能需要非常长的时间。大部分系统都会有一些应对突发情况的余量，能够吸收性能尖峰，将一些工作延迟到高峰期之后执行。但当对机器加压足够长时间之后，这些余量会被消耗尽，系统的短期尖峰也就无法维持原来的高性能。

有时候无法确认测试需要运行多长的时间才足够。如果是这样，可以让测试一直运行，持续观察直到确认系统已经稳定。下面是一个在已知系统上执行测试的例子，图 2-1 显示了系统磁盘读和写吞吐量的时序图。

43

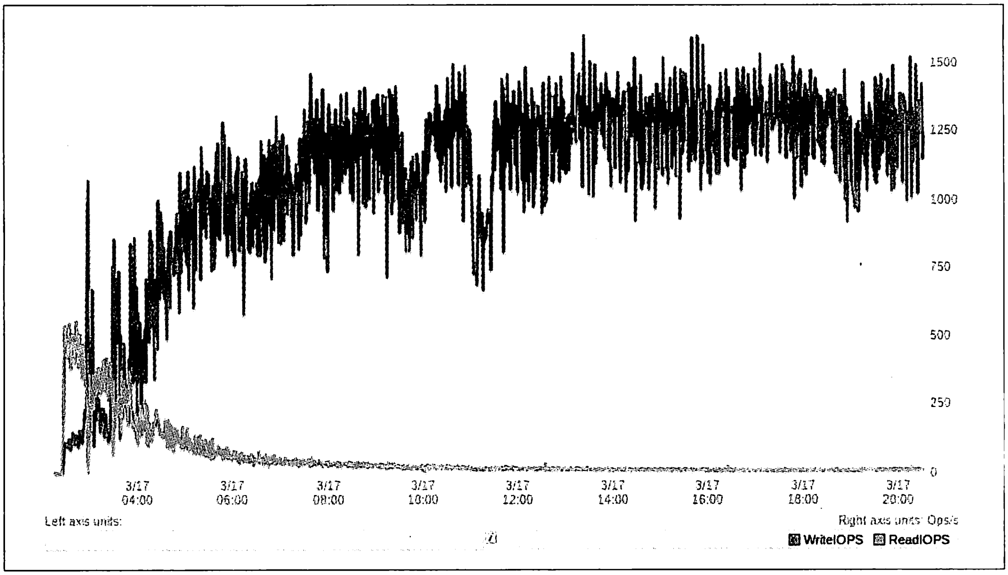


图2-1：扩展基准测试的I/O性能图

系统预热完成后，读 I/O 活动在三四个小时后曲线趋向稳定，但写 I/O 至少在八小时内变化还是很大，之后有一些点的波动较大，但读和写总体来说基本稳定了[注4]。一个简单的测试规则，就是等系统看起来稳定的时间至少等于系统预热的时间。本例中的测试持续了 72 个小时才结束，以确保能够体现系统长期的行为。

一个常见的错误的测试方式是，只执行一系列短期的测试，比如每次 60 秒，并在此测试的基础上去总结系统的性能。我们经常可以听到类似这样的话："我尝试对新版本做

注 4： 顺便说一下，写 I/O 的活动图展示的性能非常差。这个系统的稳定状态从性能上来说是一种灾难。已经达到"稳定"可以说是笑话，不过这里我们的重点在于说明系统的长期行为。

了测试，但还不如旧版本快”，然而我们分析实际的测试结果后发现，测试的方式根本不足以得出这样的结论。有时候人们也会强调说不可能有时间去测试 8 或者 12 个小时，以验证 10 个不同并发性在两到三个不同版本下的性能。如果没有时间去完成准确完整的基准测试，那么已经花费的所有时间都是一种浪费。有时候要相信别人的测试结果，这总比做一次半拉子的测试来得到一个错误的结论要好。

2.3.3 获取系统性能和状态

44

在执行基准测试时，需要尽可能多地收集被测试系统的信息。最好为基准测试建立一个目录，并且每执行一轮测试都创建单独的子目录，将测试结果、配置文件、测试指标、脚本和其他相关说明都保存在其中。即使有些结果不是目前需要的，也应该先保存下来。多余一些数据总比缺乏重要的数据要好，而且多余的数据以后也许会用得着。需要记录的数据包括系统状态和性能指标，诸如 CPU 使用率、磁盘 I/O、网络流量统计、SHOW GLOBAL STATUS 计数器等。

下面是一个收集 MySQL 测试数据的 shell 脚本：

```
#!/bin/sh

INTERVAL=5
PREFIX=$INTERVAL-sec-status
RUNFILE=/home/benchmarks/running
mysql -e 'SHOW GLOBAL VARIABLES' >> mysql-variables
while test -e $RUNFILE; do
   file=$(date +%F_%I)
   sleep=$(date +%s.%N | awk "{print $INTERVAL - (\$1 % $INTERVAL)}")
   sleep $sleep
   ts="$(date +"TS %s.%N %F %T")"
   loadavg="$(uptime)"
   echo "$ts $loadavg" >> $PREFIX-${file}-status
   mysql -e 'SHOW GLOBAL STATUS' >> $PREFIX-${file}-status &
   echo "$ts $loadavg" >> $PREFIX-${file}-innodbstatus
   mysql -e 'SHOW ENGINE INNODB STATUS\G' >> $PREFIX-${file}-innodbstatus &
   echo "$ts $loadavg" >> $PREFIX-${file}-processlist
   mysql -e 'SHOW FULL PROCESSLIST\G' >> $PREFIX-${file}-processlist &
   echo $ts
done
echo Exiting because $RUNFILE does not exist.
```

这个 shell 脚本很简单，但提供了一个有效的收集状态和性能数据的框架。看起来好像作用不大，但当需要在多个服务器上执行比较复杂的测试的时候，要回答以下关于系统行为的问题，没有这种脚本的话就会很困难了。下面是这个脚本的一些要点：

- 迭代是基于固定时间间隔的，每隔 5 秒运行一次收集的动作，注意这里 sleep 的时间有一个特殊的技巧。如果只是简单地在每次循环时插入一条“sleep 5”的指令，循

环的执行间隔时间一般都会稍大于 5 秒，那么这个脚本就没有办法通过其他脚本和图形简单地捕获时间相关的准确数据。即使有时候循环能够恰好在 5 秒内完成，但如果某些系统的时间戳是 15:32:18.218192，另外一个则是 15:32:23.819437，这时候就比较讨厌了。当然这里的 5 秒也可以改成其他的时间间隔，比如 1、10、30 或者 60 秒。不过还是推荐使用 5 秒或者 10 秒的间隔来收集数据。

45
- 每个文件名都包含了该轮测试开始的日期和小时。如果测试要持续好几天，那么这个文件可能会非常大，有必要的话需要手工将文件移到其他地方，但要分析全部结果的时候要注意从最早的文件开始。如果只需要分析某个时间点的数据，则可以根据文件名中的日期和小时迅速定位，这比在一个 GB 以上的大文件中去搜索要快捷得多。
- 每次抓取数据都会先记录当前的时间戳，所以可以在文件中搜索某个时间点的数据。也可以写一些 *awk* 或者 *sed* 脚本来简化操作。
- 这个脚本不会处理或者过滤收集到的数据。先收集所有的原始数据，然后再基于此做分析和过滤是一个好习惯。如果在收集的时候对数据做了预处理，而后续分析发现一些异常的地方需要用到更多的原始数据，这时候就要“抓瞎”了。
- 如果需要在测试完成后脚本自动退出，只需要删除 */home/benchmarks/running* 文件即可。

这只是一段简单的代码，或许不能满足全部的需求，但却很好地演示了该如何捕获测试的性能和状态数据。从代码可以看出，只捕获了 MySQL 的部分数据，如果需要，则很容易通过修改脚本添加新的数据捕获。例如，可以通过 *pt-diskstats* 工具[注5] 捕获 */proc/diskstats* 的数据为后续分析磁盘 I/O 使用。

2.3.4 获得准确的测试结果

获得准确测试结果的最好办法，是回答一些关于基准测试的基本问题：是否选择了正确的基准测试？是否为问题收集了相关的数据？是否采用了错误的测试标准？例如，是否对一个 I/O 密集型（I/O-bound）的应用，采用了 CPU 密集型（CPU-bound）的测试标准来评估性能？

接着，确认测试结果是否可重复。每次重新测试之前要确保系统的状态是一致的。如果是非常重要的测试，甚至有必要每次测试都重启系统。一般情况下，需要测试的是经过预热的系统，还需要确保预热的时间足够长（请参考前面关于基准测试需要运行多长时间的内容）、是否可重复。如果预热采用的是随机查询，那么测试结果可能就是不可重复的。

如果测试的过程会修改数据或者 schema，那么每次测试前，需要利用快照还原数据。在

注 5： 关于 *pt-diskstats* 工具的更多信息，请参考第 9 章。

表中插入 1 000 条记录和插入 100 万条记录，测试结果肯定不会相同。数据的碎片度和在磁盘上的分布，都可能导致测试是不可重复的。一个确保物理磁盘数据的分布尽可能一致的办法是，每次都进行快速格式化并进行磁盘分区复制。 46

要注意很多因素，包括外部的压力、性能分析和监控系统、详细的日志记录、周期性作业，以及其他一些因素，都会影响到测试结果。一个典型的案例，就是测试过程中突然有 *cron* 定时作业启动，或者正处于一个巡查读取周期（Patrol Read cycle），抑或 RAID 卡启动了定时的一致性检查等。要确保基准测试运行过程中所需要的资源是专用于测试的。如果有其他额外的操作，则会消耗网络带宽，或者测试基于的是和其他服务器共享的 SAN 存储，那么得到的结果很可能是不准确的。

每次测试中，修改的参数应该尽量少。如果必须要一次修改多个参数，那么可能会丢失一些信息。有些参数依赖其他参数，这些参数可能无法单独修改。有时候甚至都没有意识到这些依赖，这给测试带来了复杂性[注6]。

一般情况下，都是通过迭代逐步地修改基准测试的参数，而不是每次运行时都做大量的修改。举个例子，如果要通过调整参数来创造一个特定行为，可以通过使用分治法（divide-and-conquer，每次运行时将参数对分减半）来找到正确的值。

很多基准测试都是用来做预测系统迁移后的性能的，比如从 Oracle 迁移到 MySQL。这种测试通常比较麻烦，因为 MySQL 执行的查询类型与 Oracle 完全不同。如果想知道在 Oracle 运行得很好的应用迁移到 MySQL 以后性能如何，通常需要重新设计 MySQL 的 schema 和查询（在某些情况下，比如，建立一个跨平台的应用时，可能想知道同一条查询是如何在两个平台运行的，不过这种情况并不多见）。

另外，基于 MySQL 的默认配置的测试没有什么意义，因为默认配置是基于消耗很少内存的极小应用的。有时候可以看到一些 MySQL 和其他商业数据库产品的对比测试，结果很让人尴尬，可能就是 MySQL 采用了默认配置的缘故。让人无语的是，这样明显有误的测试结果还容易变成头条新闻。

固态存储（SSD 或者 PCI-E 卡）给基准测试带来了很大的挑战，第 9 章将进一步讨论。

最后，如果测试中出现异常结果，不要轻易当作坏数据点而丢弃。应该认真研究并找到产生这种结果的原因。测试可能会得到有价值的结果，或者一个严重的错误，抑或基准测试的设计缺陷。如果对测试结果不了解，就不要轻易公布。有一些案例表明，异常的测试结果往往都是由于很小的错误导致的，最后搞得测试无功而返[注7]。 47

注 6： 有时，这并不是问题。例如，如果正在考虑从基于 SPARC 的 Solaris 系统迁移到基于 x86 的 GNU/Linux 系统，就没有必要测试基于 x86 的 Solaris 作为中间过程。

注 7： 本书的任何一位作者都还没发生过这样的事情，仅供参考。

2.3.5 运行基准测试并分析结果

一旦准备就绪，就可以着手基准测试，收集和分析数据了。

通常来说，自动化基准测试是个好主意。这样做可以获得更精确的测试结果。因为自动化的过程可以防止测试人员偶尔遗漏某些步骤，或者误操作。另外也有助于归档整个测试过程。

自动化的方式有很多，可以是一个 Makefile 文件或者一组脚本。脚本语言可以根据需要选择：shell、PHP、Perl 等都可以。要尽可能地使所有测试过程都自动化，包括装载数据、系统预热、执行测试、记录结果等。

> 一旦设置了正确的自动化操作，基准测试将成为一步式操作。如果只是针对某些应用做一次性的快速验证测试，可能就没必要做自动化。但只要未来可能会引用到测试结果，建议都尽量地自动化。否则到时候可能就搞不清楚是如何获得这个结果的，也不记得采用了什么参数，这样就很难再通过测试重现结果了。

基准测试通常需要运行多次。具体需要运行多少次要看对结果的记分方式，以及测试的重要程度。要提高测试的准确度，就需要多运行几次。一般在测试的实践中，可以取最好的结果值，或者所有结果的平均值，抑或从五个测试结果里取最好三个值的平均值。可以根据需要更进一步精确化测试结果。还可以对结果使用统计方法，确定置信区间（confidence interval）等。不过通常来说，不会用到这种程度的确定性结果[注8]。只要测试的结果能满足目前的需求，简单地运行几轮测试，看看结果的变化就可以了。如果结果变化很大，可以再多运行几次，或者运行更长的时间，这样都可以获得更确定的结果。

获得测试结果后，还需要对结果进行分析，也就是说，要把“数字”变成“知识”。最终的目的是回答在设计测试时的问题。理想情况下，可以获得诸如“升级到 4 核 CPU 可以在保持响应时间不变的情况下获得超过 50% 的吞吐量增长”或者“增加索引可以使查
48 询更快”的结论。如果需要更加科学化，建议在测试前读读 *null hypothesis* 一书，但大部分情况下不会要求做这么严格的基准测试。

如何从数据中抽象出有意义的结果，依赖于如何收集数据。通常需要写一些脚本来分析数据，这不仅能减轻分析的工作量，而且和自动化基准测试一样可以重复运行，并易于文档化。下面是一个非常简单的 shell 脚本，演示了如何从前面的数据采集脚本采集到的数据中抽取时间维度信息。脚本的输入参数是采集到的数据文件的名字。

注 8： 如果真的需要科学可靠的结果，应该去读读关于如何设计和执行可控测试的书籍，这个已经超出了本书讨论的范畴。

```
#!/bin/sh

# This script converts SHOW GLOBAL STATUS into a tabulated format, one line
# per sample in the input, with the metrics divided by the time elapsed
# between samples.
awk '
   BEGIN {
      printf "#ts date time load QPS";
      fmt = " %.2f";
   }
   /^TS/ { # The timestamp lines begin with TS.
      ts      = substr($2, 1, index($2, ".") - 1);
      load    = NF - 2;
      diff    = ts - prev_ts;
      prev_ts = ts;
      printf "\n%s %s %s %s", ts, $3, $4, substr($load, 1, length($load)-1);
   }
   /Queries/ {
      printf fmt, ($2-Queries)/diff;
      Queries=$2
   }
   ' "$@"
```

假设该脚本名为 *analyze*，当前面的脚本生成状态文件以后，就可以运行该脚本，可能会得到如下的结果：

```
[baron@ginger ~]$ ./analyze 5-sec-status-2011-03-20
#ts date time load QPS
1300642150 2011-03-20 17:29:10 0.00 0.62
1300642155 2011-03-20 17:29:15 0.00 1311.60
1300642160 2011-03-20 17:29:20 0.00 1770.60
1300642165 2011-03-20 17:29:25 0.00 1756.60
1300642170 2011-03-20 17:29:30 0.00 1752.40
1300642175 2011-03-20 17:29:35 0.00 1735.00
1300642180 2011-03-20 17:29:40 0.00 1713.00
1300642185 2011-03-20 17:29:45 0.00 1788.00
1300642190 2011-03-20 17:29:50 0.00 1596.40
```

第一行是列的名字；第二行的数据应该忽略，因为这是测试实际启动前的数据。接下来的行包含 Unix 时间戳、日期、时间（注意时间数据是每 5 秒更新一次，前面脚本说明时曾提过）、系统负载、数据库的 QPS（每秒查询次数）五列，这应该是用于分析系统性能的最少数据需求了。接下来将演示如何根据这些数据快速地绘成图形，并分析基准测试过程中发生了什么。 49

2.3.6 绘图的重要性

如果你想要统治世界，就必须不断地利用“阴谋”[注9]。而最简单有效的图形，就是将性能指标按照时间顺序绘制。通过图形可以立刻发现一些问题，而这些问题在原始数据中却

注 9： 英语中 plot 既有“阴谋”的意思，也有“绘图”的意思，所以这里是一句双关语。——译者注

很难被注意到。或许你会坚持看测试工具打印出来的平均值或其他汇总过的信息，但平均值有时候是没有用的，它会掩盖掉一些真实情况。幸运的是，前面写的脚本的输出都可以定制作为 *gnuplot* 或者 *R* 绘图的数据来源。假设使用 *gnuplot*，假设输出的数据文件名是 *QPS-per-5-seconds*：

```
gnuplot> plot "QPS-per-5-seconds" using 5 w lines title "QPS"
```

该 *gnuplot* 命令将文件的第五列 *qps* 数据绘成图形，图的标题是 QPS。图 2-2 是绘制出来的结果图。

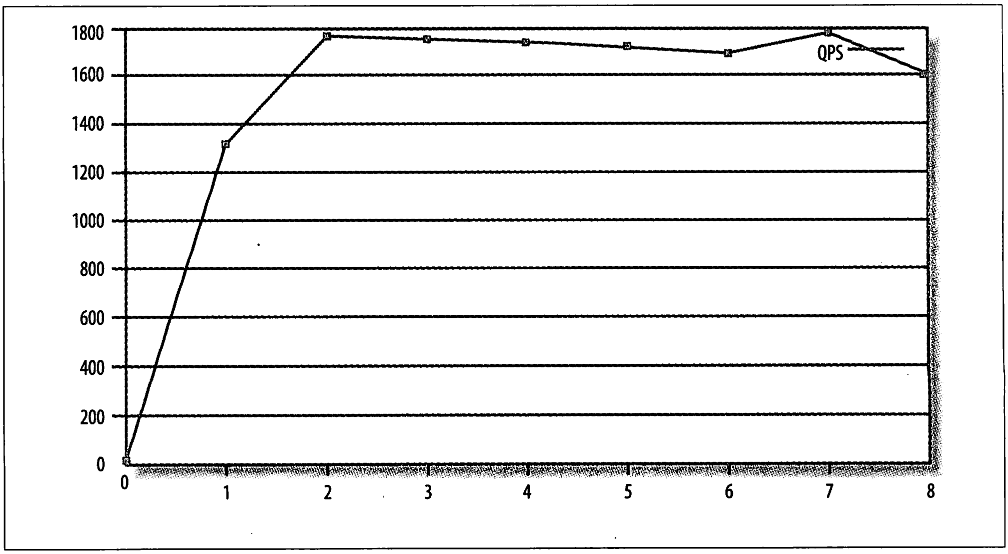


图2-2：基准测试的QPS图形

下面我们讨论一个可以更加体现图形价值的例子。假设 MySQL 数据正在遭受"疯狂刷新（furious flushing）"的问题，在刷新落后于检查点时会阻塞所有的活动，从而导致吞吐量严重下跌。95% 的响应时间和平均响应时间指标都无法发现这个问题，也就是说这
50 两个指标掩盖了问题。但图形会显示出这个周期性的问题，请参考图 2-3。

图 2-3 显示的是每分钟新订单的交易量（NOTPM，new-order transactions per minute）。从曲线可以看到明显的周期性下降，但如果从平均值（点状虚线）来看波动很小。一开始的低谷是由于系统的缓存是空的，而后面其他的下跌则是由于系统刷新脏块到磁盘导致。如果没有图形，要发现这个趋势会比较困难。

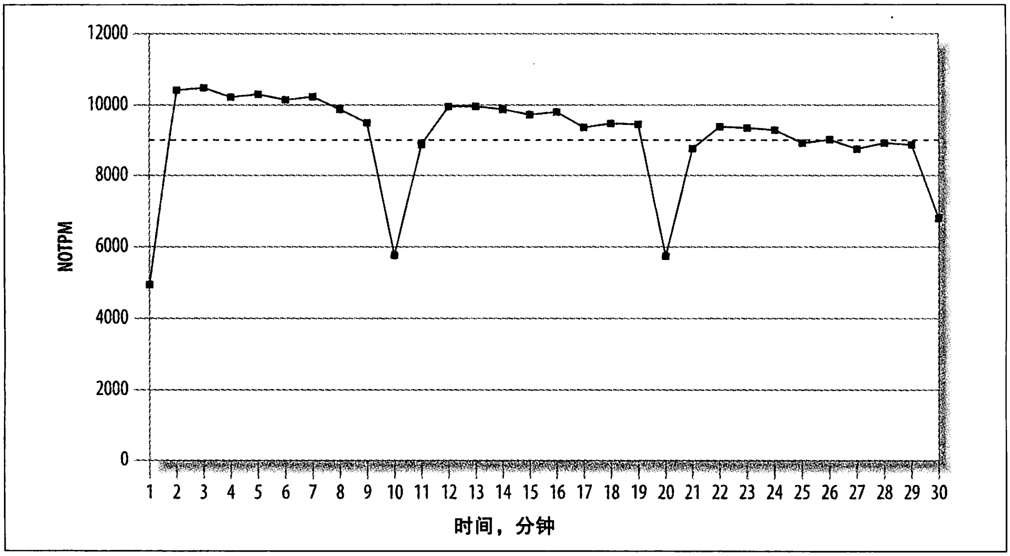


图2-3：一个30分钟的*dbt2*测试的结果

这种性能尖刺在压力大的系统比较常见，需要调查原因。在这个案例中，是由于使用了旧版本的 InnoDB 引擎，脏块的刷新算法性能很差。但这个结论不能是想当然的，需要认真地分析详细的性能统计。在性能下跌时，`SHOW ENGINE INNODB STATUS`的输出是什么？`SHOW FULL PROCESSLIST`的输出是什么？应该可以发现 InnoDB 在持续地刷新脏块，并且阻塞了很多状态是"waiting on query cache lock"的线程，或者其他类似的现象。在执行基准测试的时候要尽可能地收集更多的细节数据，然后将数据绘制成图形，这样可以帮助快速地发现问题。

2.4 基准测试工具

没有必要开发自己的基准测试系统，除非现有的工具确实无法满足需求。下面的章节会介绍一些可用的工具。

2.4.1 集成式测试工具

51

回忆一下前文提供的两种测试类型：集成式测试和单组件式测试。毫不奇怪，有些工具是针对整个应用进行测试，也有些工具是针对 MySQL 或者其他组件单独进行测试的。集成式测试，通常是获得整个应用概况的最佳手段。已有的集成式测试工具如下所示。

ab

ab 是一个 Apache HTTP 服务器基准测试工具。它可以测试 HTTP 服务器每秒最多

可以处理多少请求。如果测试的是 Web 应用服务，这个结果可以转换成整个应用每秒可以满足多少请求。这是个非常简单的工具，用途也有限，只能针对单个 URL 进行尽可能快的压力测试。关于 *ab* 的更多信息可以参考 *http://httpd.apache.org/docs/2.0/programs/ab.html*。

http_load

这个工具概念上和 *ab* 类似，也被设计为对 Web 服务器进行测试，但比 *ab* 要更加灵活。可以通过一个输入文件提供多个 URL，*http_load* 在这些 URL 中随机选择进行测试。也可以定制 *http_load*，使其按照时间比率进行测试，而不仅仅是测试最大请求处理能力。更多信息请参考 *http://www.acme.com/software/http-load/*。

JMeter

JMeter 是一个 Java 应用程序，可以加载其他应用并测试其性能。它虽然是设计用来测试 Web 应用的，但也可以用于测试其他诸如 FTP 服务器，或者通过 JDBC 进行数据库查询测试。

JMeter 比 *ab* 和 *http_load* 都要复杂得多。例如，它可以通过控制预热时间等参数，更加灵活地模拟真实用户的访问。JMeter 拥有绘图接口（带有内置的图形化处理的功能），还可以对测试进行记录，然后离线重演测试结果。更多信息请参考 *http://jakarta.apache.org/jmeter/*。

2.4.2 单组件式测试工具

有一些有用的工具可以测试 MySQL 和基于 MySQL 的系统的性能。2.5 节将演示如何利用这些工具进行测试。

mysqlslap

mysqlslap（*http://dev.mysql.com/doc/refman/5.1/en/mysqlslap.html*）可以模拟服务器的负载，并输出计时信息。它包含在 MySQL 5.1 的发行包中，应该在 MySQL 4.1 或者更新的版本中都可以使用。测试时可以执行并发连接数，并指定 SQL 语句（可以在命令行上执行，也可以把 SQL 语句写入到参数文件中）。如果没有指定 SQL 语句，*mysqlslap* 会自动生成查询 schema 的 SELECT 语句。

52 *MySQL Benchmark Suite (sql-bench)*

在 MySQL 的发行包中也提供了一款自己的基准测试套件，可以用于在不同数据库服务器上进行比较测试。它是单线程的，主要用于测试服务器执行查询的速度。结果会显示哪种类型的操作在服务器上执行得更快。

这个测试套件的主要好处是包含了大量预定义的测试，容易使用，所以可以很轻松地用于比较不同存储引擎或者不同配置的性能测试。其也可以用于高层次测试，比较两个服务器的总体性能。当然也可以只执行预定义测试的子集（例如只测试

UPDATE 的性能）。这些测试大部分是 CPU 密集型的，但也有些短时间的测试需要大量的磁盘 I/O 操作。

这个套件的最大缺点主要有：它是单用户模式的，测试的数据集很小且用户无法使用指定的数据，并且同一个测试多次运行的结果可能会相差很大。因为是单线程且串行执行的，所以无法测试多 CPU 的能力，只能用于比较单 CPU 服务器的性能差别。使用这个套件测试数据库服务器还需要 Perl 和 BDB 的支持，相关文档请参考 *http://dev.mysql.com/doc/en/mysql-benchmarks.html/*。

Super Smack

Super Smack（*http://vegan.net/tony/supersmack/*）是一款用于 MySQL 和 PostgreSQL 的基准测试工具，可以提供压力测试和负载生成。这是一个复杂而强大的工具，可以模拟多用户访问，可以加载测试数据到数据库，并支持使用随机数据填充测试表。测试定义在“smack”文件中，smack 文件使用一种简单的语法定义测试的客户端、表、查询等测试要素。

Database Test Suite

Database Test Suite 是由开源软件开发实验室（OSDL，Open Source Development Labs）设计的，发布在 SourceForge 网站（*http://sourceforge.net/projects/osdldbt/*）上，这是一款类似某些工业标准测试的测试工具集，例如由事务处理性能委员会（TPC，Transaction Processing Performance Council）制定的各种标准。特别值得一提的是，其中的 *dbt2* 就是一款免费的 TPC-C OLTP 测试工具（未认证）。之前本书作者经常使用该工具，不过现在已经使用自己研发的专用于 MySQL 的测试工具替代了。

Percona's TPCC-MySQL Tool

我们开发了一个类似 TPC-C 的基准测试工具集，其中有部分是专门为 MySQL 测试开发的。在评估大压力下 MySQL 的一些行为时，我们经常会利用这个工具进行测试（简单的测试，一般会采用 *sysbench* 替代）。该工具的源代码可以在 *https://launchpad.net/perconatools* 下载，在源码库中有一个简单的文档说明。

sysbench 53

sysbench（*https://launchpad.net/sysbench*）是一款多线程系统压测工具。它可以根据影响数据库服务器性能的各种因素来评估系统的性能。例如，可以用来测试文件 I/O、操作系统调度器、内存分配和传输速度、POSIX 线程，以及数据库服务器等。*sysbench* 支持 Lua 脚本语言（*http://www.lua.org*），Lua 对于各种测试场景的设置可以非常灵活。*sysbench* 是我们非常喜欢的一种全能测试工具，支持 MySQL、操作系统和硬件的硬件测试。

MySQL 的 BENCHMARK() 函数

MySQL 有一个内置的 BENCHMARK() 函数，可以测试某些特定操作的执行速度。参数可以是需要执行的次数和表达式。表达式可以是任何的标量表达式，比如返回值是标量的子查询或者函数。该函数可以很方便地测试某些特定操作的性能，比如通过测试可以发现，MD5() 函数比 SHA1() 函数要快：

```
mysql> SET @input := 'hello world';
mysql> SELECT BENCHMARK(1000000, MD5(@input));
+---------------------------------+
| BENCHMARK(1000000, MD5(@input)) |
+---------------------------------+
|                               0 |
+---------------------------------+
1 row in set (2.78 sec)
mysql> SELECT BENCHMARK(1000000, SHA1(@input));
+----------------------------------+
| BENCHMARK(1000000, SHA1(@input)) |
+----------------------------------+
|                                0 |
+----------------------------------+
1 row in set (3.50 sec)
```

执行后的返回值永远是 0，但可以通过客户端返回的时间来判断执行的时间。在这个例子中可以看到 MD5() 执行比 SHA1() 要快。使用 BENCHMARK() 函数来测试性能，需要清楚地知道其原理，否则容易误用。这个函数只是简单地返回服务器执行表达式的时间，而不会涉及分析和优化的开销。而且表达式必须像这个例子一样包含用户定义的变量，否则多次执行同样的表达式会因为系统缓存命中而影响结果[注 10]。

虽然 BENCHMARK() 函数用起来很方便，但不合适用来做真正的基准测试，因为很难理解真正要测试的是什么，而且测试的只是整个执行周期中的一部分环节。

54

2.5 基准测试案例

本节将演示一些利用上面提到的基准测试工具进行测试的真实案例。这些案例未必涵盖所有测试工具，但应该可以帮助读者针对自己的测试需要来做出判断和选择，并作为入门的开端。

注 10： 本书作者之一碰到了这个问题，因为发现循环执行 1 000 次表达式和只执行一次表达式的时间居然差不多，这只能说明缓存命中了。实际上，当碰到此类情况时，第一反应就应当是缓存命中或者出错了。

2.5.1 http_load

下面通过一个简单的例子来演示如何使用 *http_load*。首先创建一个 *urls.txt* 文件，输入如下的 URL：

```
http://www.mysqlperformanceblog.com/
http://www.mysqlperformanceblog.com/page/2/
http://www.mysqlperformanceblog.com/mysql-patches/
http://www.mysqlperformanceblog.com/mysql-performance-presentations/
http://www.mysqlperformanceblog.com/2006/09/06/slow-query-log-analyzes-tools/
```

http_load 最简单的用法，就是循环请求给定的 URL 列表。测试程序将以最快的速度请求这些 URL：

```
$ http_load -parallel 1 -seconds 10 urls.txt
19 fetches, 1 max parallel, 837929 bytes, in 10.0003 seconds
44101.5 mean bytes/connection
1.89995 fetches/sec, 83790.7 bytes/sec
msecs/connect: 41.6647 mean, 56.156 max, 38.21 min
msecs/first-response: 320.207 mean, 508.958 max, 179.308 min
HTTP response codes:
  code 200 - 19
```

测试的结果很容易理解，只是简单地输出了请求的统计信息。下面是另外一个稍微复杂的测试，还是尽可能快地循环请求给定的 URL 列表，不过模拟同时有五个并发用户在进行请求：

```
$ http_load -parallel 5 -seconds 10 urls.txt
94 fetches, 5 max parallel, 4.75565e+06 bytes, in 10.0005 seconds
50592 mean bytes/connection
9.39953 fetches/sec, 475541 bytes/sec
msecs/connect: 65.1983 mean, 169.991 max, 38.189 min
msecs/first-response: 245.014 mean, 993.059 max, 99.646 min
HTTP response codes:
  code 200 - 94
```

另外，除了测试最快的速度，也可以根据预估的访问请求率（比如每秒 5 次）来做压力模拟测试。

```
$ http_load -rate 5 -seconds 10 urls.txt
48 fetches, 4 max parallel, 2.50104e+06 bytes, in 10 seconds
52105 mean bytes/connection
4.8 fetches/sec, 250104 bytes/sec
msecs/connect: 42.5931 mean, 60.462 max, 38.117 min
msecs/first-response: 246.811 mean, 546.203 max, 108.363 min
HTTP response codes:
  code 200 - 48
```

55

最后，还可以模拟更大的负载，可以将访问请求率提高到每秒 20 次请求。请注意，连接和请求响应时间都会随着负载的提高而增加。

```
$ http_load -rate 20 -seconds 10 urls.txt
111 fetches, 89 max parallel, 5.91142e+06 bytes, in 10.0001 seconds
53256.1 mean bytes/connection
11.0998 fetches/sec, 591134 bytes/sec
msecs/connect: 100.384 mean, 211.885 max, 38.214 min
msecs/first-response: 2163.51 mean, 7862.77 max, 933.708 min
HTTP response codes:
  code 200 -- 111
```

2.5.2 MySQL 基准测试套件

MySQL 基准测试套件（MySQL Benchmark Suite）由一组基于 Perl 开发的基准测试工具组成。在 MySQL 安装目录下的 *sql-bench* 子目录中包含了该工具。比如在 Debian GNU/Linux 系统上，默认的路径是 */usr/share/mysql/sql-bench*。

在用这个工具集测试前，应该读一下 *README* 文件，了解使用方法和命令行参数说明。如果要运行全部测试，可以使用如下的命令：

```
$ cd /usr/share/mysql/sql-bench/
sql-bench$ ./run-all-tests --server=mysql --user=root --log --fast
Test finished. You can find the result in:
output/RUN-mysql_fast-Linux_2.4.18_686_smp_i686
```

运行全部测试需要比较长的时间，有可能会超过一个小时，其具体长短依赖于测试的硬件环境和配置。如果指定了 *--log* 命令行，则可以监控到测试的进度。测试的结果都保存在 *output* 子目录中，每项测试的结果文件中都会包含一系列的操作计时信息。下面是一个具体的例子，为方便印刷，部分格式做了修改。

```
sql-bench$ tail -5 output/select-mysql_fast-Linux_2.4.18_686_smp_i686
Time for count_distinct_group_on_key (1000:6000):
  34  wallclock secs ( 0.20 usr  0.08 sys +  0.00 cusr  0.00 csys =  0.28 CPU)
Time for count_distinct_group_on_key_parts (1000:100000):
  34  wallclock secs ( 0.57 usr  0.27 sys +  0.00 cusr  0.00 csys =  0.84 CPU)
Time for count_distinct_group (1000:100000):
  34  wallclock secs ( 0.59 usr  0.20 sys +  0.00 cusr  0.00 csys =  0.79 CPU)
Time for count_distinct_big (100:1000000):
  8   wallclock secs ( 4.22 usr  2.20 sys +  0.00 cusr  0.00 csys =  6.42 CPU)
Total time:
  868 wallclock secs (33.24 usr  9.55 sys +  0.00 cusr  0.00 csys = 42.79 CPU)
```

如上所示，count_distinct_group_on_key(1000:6000) 测试花费了 34 秒（wallclock secs），这是客户端运行测试花费的总时间；其他值（包括 usr，sys，cursr，csys）则占了测
56 试的 0.28 秒的开销，这是运行客户端测试代码所花费的时间，而不是等待 MySQL 服务器响应的时间。而测试者真正需要关心的测试结果，是除去客户端控制的部分，即实际运行时间应该是 33.72 秒。

除了运行全部测试集外，也可以选择单独执行其中的部分测试项。例如可以选择只执行

insert 测试，这会比运行全部测试集所得到的汇总信息给出更多的详细信息：

```
sql-bench$ ./test-insert
Testing server 'MySQL 4.0.13 log' at 2003-05-18 11:02:39

Testing the speed of inserting data into 1 table and do some selects on it.
The tests are done with a table that has 100000 rows.

Generating random keys
Creating tables
Inserting 100000 rows in order
Inserting 100000 rows in reverse order
Inserting 100000 rows in random order
Time for insert (300000):
  42 wallclock secs ( 7.91 usr  5.03 sys +  0.00 cusr  0.00 csys = 12.94 CPU)
Testing insert of duplicates
Time for insert_duplicates (100000):
  16 wallclock secs ( 2.28 usr  1.89 sys +  0.00 cusr  0.00 csys =  4.17 CPU)
```

2.5.3 sysbench

sysbench 可以执行多种类型的基准测试，它不仅设计用来测试数据库的性能，也可以测试运行数据库的服务器的性能。实际上，Peter 和 Vadim 最初设计这个工具是用来执行 MySQL 性能测试的（尽管并不能完成所有的 MySQL 基准测试）。下面先演示一些非 MySQL 的测试场景，来测试各个子系统的性能，这些测试可以用来评估系统的整体性能瓶颈。后面再演示如何测试数据库的性能。

强烈建议大家都能熟悉 *sysbench* 测试，在 MySQL 用户的工具包中，这应该是最有用的工具之一。尽管有其他很多测试工具可以替代 *sysbench* 的某些功能，但那些工具有时候并不可靠，获得的结果也不一定和 MySQL 性能相关。例如，I/O 性能测试可以用 *iozone*、*bonnie++* 等一系列工具，但需要注意设计场景，以便可以模拟 InnoDB 的磁盘 I/O 模式。而 *sysbench* 的 I/O 测试则和 InnoDB 的 I/O 模式非常类似，所以 fileio 选项是非常好用的。

sysbench 的 CPU 基准测试

最典型的子系统测试就是 CPU 基准测试。该测试使用 64 位整数，测试计算素数直到某个最大值所需要的时间。下面的例子将比较两台不同的 GNU/Linux 服务器上的测试结果。第一台机器的 CPU 配置如下：

57

```
[server1 ~]$ cat /proc/cpuinfo
...
model name      : AMD Opteron(tm) Processor 246
stepping        : 1
cpu MHz         : 1992.857
cache size      : 1024 KB
```

在这台服务器上运行如下的测试：

```
[server1 ~]$ sysbench --test=cpu --cpu-max-prime=20000 run
sysbench v0.4.8:  multithreaded system evaluation benchmark
...
Test execution summary:    total time:                        121.7404s
```

第二台服务器配置了不同的 CPU：

```
[server2 ~]$ cat /proc/cpuinfo
...
model name      : Intel(R) Xeon(R) CPU           5130  @ 2.00GHz
stepping        : 6
cpu MHz         : 1995.005
```

测试结果如下：

```
[server1 ~]$ sysbench --test=cpu --cpu-max-prime=20000 run
sysbench v0.4.8:  multithreaded system evaluation benchmark
...
Test execution summary:    total time:                        61.8596s
```

测试的结果简单打印出了计算出素数的时间，很容易进行比较。在上面的测试中，第二台服务器的测试结果显示比第一台快两倍。

sysbench 的文件 I/O 基准测试

文件 I/O（`fileio`）基准测试可以测试系统在不同 I/O 负载下的性能。这对于比较不同的硬盘驱动器、不同的 RAID 卡、不同的 RAID 模式，都很有帮助。可以根据测试结果来调整 I/O 子系统。文件 I/O 基准测试模拟了很多 InnoDB 的 I/O 特性。

测试的第一步是准备（prepare）阶段，生成测试用到的数据文件，生成的数据文件至少要比内存大。如果文件中的数据能完全放入内存中，则操作系统缓存大部分的数据，导致测试结果无法体现 I/O 密集型的工作负载。首先通过下面的命令创建一个数据集：

```
$ sysbench --test=fileio --file-total-size=150G prepare
```

这个命令会在当前工作目录下创建测试文件，后续的运行（run）阶段将通过读写这些文件进行测试。第二步就是运行（run）阶段，针对不同的 I/O 类型有不同的测试选项：

58 seqwr

顺序写入。

seqrewr

顺序重写。

seqrd

　　顺序读取。

rndrd

　　随机读取。

rndwr

　　随机写入。

rdnrw

　　混合随机读 / 写。

下面的命令运行文件 I/O 混合随机读 / 写基准测试：

```
$ sysbench --test=fileio --file-total-size=150G --file-test-mode=rndrw/
--init-rng=on --max-time=300 --max-requests=0 run
```

结果如下：

```
sysbench v0.4.8:  multithreaded system evaluation benchmark

Running the test with following options:
Number of threads: 1
Initializing random number generator from timer.

Extra file open flags: 0
128 files, 1.1719Gb each
150Gb total file size
Block size 16Kb
Number of random requests for random IO: 10000
Read/Write ratio for combined random IO test: 1.50
Periodic FSYNC enabled, calling fsync() each 100 requests.
Calling fsync() at the end of test, Enabled.
Using synchronous I/O mode
Doing random r/w test
Threads started!
Time limit exceeded, exiting...
Done.

Operations performed:  40260 Read, 26840 Write, 85785 Other = 152885 Total
Read 629.06Mb  Written 419.38Mb  Total transferred 1.0239Gb  (3.4948Mb/sec)
  223.67 Requests/sec executed

Test execution summary:
    total time:                          300.0004s
    total number of events:              67100
    total time taken by event execution: 254.4601
    per-request statistics:
         min:                            0.0000s
         avg:                            0.0038s
```

59

```
        max:                              0.5628s
        approx. 95 percentile:            0.0099s

Threads fairness:
    events (avg/stddev):           67100.0000/0.00
    execution time (avg/stddev):   254.4601/0.00
```

输出结果中包含了大量的信息。和 I/O 子系统密切相关的包括每秒请求数和总吞吐量。在上述例子中，每秒请求数是 223.67 Requests/sec，吞吐量是 3.4948MB/sec。另外，时间信息也非常有用，尤其是大约 95% 的时间分布。这些数据对于评估磁盘性能十分有用。

测试完成后，运行清除（cleanup）操作删除第一步生成的测试文件：

```
$ sysbench --test=fileio --file-total-size=150G cleanup
```

sysbench 的 OLTP 基准测试

OLTP 基准测试模拟了一个简单的事务处理系统的工作负载。下面的例子使用的是一张超过百万行记录的表，第一步是先生成这张表：

```
$ sysbench --test=oltp --oltp-table-size=1000000 --mysql-db=test/
--mysql-user=root prepare
sysbench v0.4.8:  multithreaded system evaluation benchmark

No DB drivers specified, using mysql
Creating table 'sbtest'...
Creating 1000000 records in table 'sbtest'...
```

生成测试数据只需要上面这条简单的命令即可。接下来可以运行测试，这个例子采用了 8 个并发线程，只读模式，测试时长 60 秒：

```
$ sysbench --test=oltp --oltp-table-size=1000000 --mysql-db=test --mysql-user=root/
--max-time=60 --oltp-read-only=on --max-requests=0 --num-threads=8 run
sysbench v0.4.8:  multithreaded system evaluation benchmark

No DB drivers specified, using mysql
WARNING: Preparing of "BEGIN" is unsupported, using emulation
(last message repeated 7 times)
Running the test with following options:
Number of threads: 8

Doing OLTP test.
Running mixed OLTP test
Doing read-only test
Using Special distribution (12 iterations,  1 pct of values are returned in 75 pct
cases)
Using "BEGIN" for starting transactions
Using auto_inc on the id column
Threads started!
Time limit exceeded, exiting...
```

```
(last message repeated 7 times)
Done.

OLTP test statistics:
    queries performed:
        read:                            179606
        write:                           0
        other:                           25658
        total:                           205264
    transactions:                        12829  (213.07 per sec.)
    deadlocks:                           0      (0.00 per sec.)
    read/write requests:                 179606 (2982.92 per sec.)
    other operations:                    25658  (426.13 per sec.)

Test execution summary:
    total time:                          60.2114s
    total number of events:              12829
    total time taken by event execution: 480.2086

    per-request statistics:
         min:                            0.0030s
         avg:                            0.0374s
         max:                            1.9106s
         approx. 95 percentile:         0.1163s

Threads fairness:
    events (avg/stddev):           1603.6250/70.66
    execution time (avg/stddev):   60.0261/0.06
```

如上所示，结果中包含了相当多的信息。其中最有价值的信息如下：

- 总的事务数。
- 每秒事务数。
- 时间统计信息（最小、平均、最大响应时间，以及 95% 百分比响应时间）。
- 线程公平性统计信息（thread-fairness），用于表示模拟负载的公平性。

这个例子使用的是 *sysbench* 的第 4 版，在 SourceForge.net 可以下载到这个版本的编译好的可执行文件。也可以从 Launchpad 下载最新的第 5 版的源代码自行编译（这是一件简单、有用的事情），这样就可以利用很多新版本的特性，包括可以基于多个表而不是单个表进行测试，可以每隔一定的间隔比如 10 秒打印出吞吐量和响应的结果。这些指标对于理解系统的行为非常重要。

sysbench 的其他特性

sysbench 还有一些其他的基准测试，但和数据库性能没有直接关系。

内存（memory）

测试内存的连续读写性能。

61 *线程*（thread）

测试线程调度器的性能。对于高负载情况下测试线程调度器的行为非常有用。

互斥锁（mutex）

测试互斥锁（mutex）的性能，方式是模拟所有线程在同一时刻并发运行，并都短暂请求互斥锁（互斥锁 mutex 是一种数据结构，用来对某些资源进行排他性访问控制，防止因并发访问导致问题）。

顺序写（seqwr）

测试顺序写的性能。这对于测试系统的实际性能瓶颈很重要。可以用来测试 RAID 控制器的高速缓存的性能状况，如果测试结果异常则需要引起重视。例如，如果 RAID 控制器写缓存没有电池保护，而磁盘的压力达到了 3 000 次请求 / 秒，就是一个问题，数据可能是不安全的。

另外，除了指定测试模式参数（*--test*）外，*sysbench* 还有其他很多参数，比如 *--num-threads*、*--max-requests* 和 *--max-time* 参数，更多信息请查阅相关文档。

2.5.4 数据库测试套件中的 dbt2 TPC-C 测试

数据库测试套件（Database Test Suite）中的 *dbt2* 是一款免费的 TPC-C 测试工具。TPC-C 是 TPC 组织发布的一个测试规范，用于模拟测试复杂的在线事务处理系统（OLTP）。它的测试结果包括每分钟事务数（tpmC），以及每事务的成本（Price/tpmC）。这种测试的结果非常依赖硬件环境，所以公开发布的 TPC-C 测试结果都会包含具体的系统硬件配置信息。

> *dbt2* 并不是真正的 TPC-C 测试，它没有得到 TPC 组织的认证，它的结果不能直接跟 TPC-C 的结果做对比。而且本书作者开发了一款比 *dbt2* 更好的测试工具，详细情况见 2.5.5 节。

下面看一个设置和运行 *dbt2* 基准测试的例子。这里使用的是 *dbt2* 0.37 版本，这个版本能够支持 MySQL 的最新版本（还有更新的版本，但包含了一些 MySQL 不能提供完全支持的修正）。下面是测试步骤。

1. 准备测试数据。

   下面的命令会在指定的目录创建用于 10 个仓库的数据。每个仓库使用大约 700MB 磁盘空间，测试所需要的总的磁盘空间和仓库的数量成正比。因此，可以通过 *-w* 参数来调整仓库的个数以生成合适大小的数据集。

```
# src/datagen -w 10 -d /mnt/data/dbt2-w10
warehouses = 10
districts = 10
customers = 3000
items = 100000
orders = 3000
stock = 100000
new_orders = 900

Output directory of data files: /mnt/data/dbt2-w10

Generating data files for 10 warehouse(s)...
Generating item table data...
Finished item table data...
Generating warehouse table data...
Finished warehouse table data...
Generating stock table data...
```

2. 加载数据到 MySQL 数据库。

下面的命令创建一个名为 dbt2w10 的数据库，并且将上一步生成的测试数据加载到数据库中（*-d* 参数指定数据库，*-f* 参数指定测试数据所在的目录）。

```
# scripts/mysql/mysql_load_db.sh -d dbt2w10 -f /mnt/data/dbt2-w10/
-s /var/lib/mysql/mysql.sock
```

3. 运行测试。

最后一步是运行 *scripts* 脚本目录中的如下命令执行测试：

```
# run_mysql.sh -c 10 -w 10 -t 300 -n dbt2w10/
-u root -o /var/lib/mysql/mysql.sock-e
************************************************************************
*                     DBT2 test for MySQL  started                     *
*                                                                      *
*            Results can be found in output/9 directory                *
************************************************************************
*                                                                      *
*  Test consists of 4 stages:                                          *
*                                                                      *
*  1. Start of client to create pool of databases connections          *
*  2. Start of driver to emulate terminals and transactions generation *
*  3. Test                                                             *
*  4. Processing of results                                            *
*                                                                      *
************************************************************************

DATABASE NAME:                dbt2w10
DATABASE USER:                root
DATABASE SOCKET:              /var/lib/mysql/mysql.sock
DATABASE CONNECTIONS:         10
TERMINAL THREADS:             100
SCALE FACTOR(WARHOUSES):      10
TERMINALS PER WAREHOUSE:      10
DURATION OF TEST(in sec):     300
SLEEPY in (msec)              300
```

63

```
ZERO DELAYS MODE:            1

Stage 1. Starting up client...
Delay for each thread - 300 msec. Will sleep for 4 sec to start 10 database
connections
CLIENT_PID = 12962

Stage 2. Starting up driver...
Delay for each thread - 300 msec. Will sleep for 34 sec to start 100 terminal
threads
All threads has spawned successfuly.

Stage 3. Starting of the test. Duration of the test 300 sec

Stage 4. Processing of results...
Shutdown clients. Send TERM signal to 12962.
 Response Time (s)
 Transaction      %  Average :  90th %   Total  Rollbacks      %
------------  -----  -----------------  ------  ---------  -----
    Delivery   3.53    2.224 :   3.059    1603          0   0.00
   New Order  41.24    0.659 :   1.175   18742        172   0.92
Order Status   3.86    0.684 :   1.228    1756          0   0.00
     Payment  39.23    0.644 :   1.161   17827          0   0.00
 Stock Level   3.59    0.652 :   1.147    1630          0   0.00

3396.95 new-order transactions per minute (NOTPM)
5.5 minute duration
0 total unknown errors
31 second(s) ramping up
```

最重要的结果是输出信息中末尾处的一行：

```
3396.95 new-order transactions per minute (NOTPM)
```

这里显示了系统每分钟可以处理的最大事务数，越大越好（new-order 并非一种事务类型的专用术语，它只是表明测试是模拟用户在假想的电子商务网站下的新订单）。

通过修改某些参数可以定制不同的基准测试。

-c

到数据库的连接数。修改该参数可以模拟不同程度的并发性，以测试系统的可扩展性。

-e

启用零延迟（zero-delay）模式，这意味着在不同查询之间没有时间延迟。这可以对数据库施加更大的压力，但不符合真实情况。因为真实的用户在执行一个新查询前总需要一个“思考时间（think time）”。

-t

基准测试的持续时间。这个参数应该精心设置，否则可能导致测试的结果是无意义的。对于 I/O 密集型的基准测试，太短的持续时间会导致错误的结果，因为系统可能还

没有足够的时间对缓存进行预热。而对于 CPU 密集型的基准测试，这个时间又不应该设置得太长；否则生成的数据量过大，可能转变成 I/O 密集型。 64

这种基准测试的结果，可以比单纯的性能测试提供更多的信息。例如，如果发现测试有很多的回滚现象，那么就可以判定很可能什么地方出现错误了。

2.5.5 Percona 的 TPCC-MySQL 测试工具

尽管 *sysbench* 的测试很简单，并且结果也具有可比性，但毕竟无法模拟真实的业务压力。相比而言，TPC-C 测试则能模拟真实压力。2.5.4 节谈到的 *dbt2* 是 TPC-C 的一个很好的实现，但也还有一些不足之处。为了满足很多大型基准测试的需求，本书的作者重新开发了一款新的类 TPC-C 测试工具，代码放在 Launchpad 上，可以通过如下地址获取：*https://code.launchpad.net/~percona-dev/perconatools/tpcc-mysql*，其中包含了一个 *README* 文件说明了如何编译。该工具使用很简单，但测试数据中的仓库数量很多，可能需要用到其中的并行数据加载工具来加快准备测试数据集的速度，否则这一步会花费很长时间。

使用这个测试工具，需要创建数据库和表结构、加载数据、执行测试三个步骤。数据库和表结构通过包含在源码中的 SQL 脚本创建。加载数据通过用 C 写的 *tpcc_load* 工具完成，该工具需要自行编译。加载数据需要执行一段时间，并且会产生大量的输出信息（一般都应该将程序输出重定向到文件中，这里尤其应该如此，否则可能丢失滚动的历史信息）。下面的例子显示了配置过程，创建了一个小型（五个仓库）的测试数据集，数据库名为 tpcc5。

```
$ ./tpcc_load localhost tpcc5 username p4ssword 5
*************************************
*** ###easy### TPC-C Data Loader  ***
*************************************
<Parameters>
     [server]: localhost
     [port]: 3306
     [DBname]: tpcc5
       [user]: username
       [pass]: p4ssword
  [warehouse]: 5
TPCC Data Load Started...
Loading Item
.................................................. 5000
.................................................. 10000
.................................................. 15000

[output snipped for brevity]

Loading Orders for D=10, W= 5
.......... 1000
```

65

```
.......... 2000
.......... 3000
Orders Done.

...DATA LOADING COMPLETED SUCCESSFULLY.
```

然后，使用 *tpcc_start* 工具开始执行基准测试。其同样会产生很多输出信息，还是建议重定向到文件中。下面是一个简单的示例，使用五个线程操作五个仓库，30 秒预热时间，30 秒测试时间：

```
$ ./tpcc_start localhost tpcc5 username p4ssword 5 5 30 30
***************************************
*** ###easy### TPC-C Load Generator ***
***************************************
<Parameters>
     [server]: localhost
     [port]: 3306
     [DBname]: tpcc5
       [user]: username
       [pass]: p4ssword
  [warehouse]: 5
 [connection]: 5
     [rampup]: 30 (sec.)
    [measure]: 30 (sec.)

RAMP-UP TIME.(30 sec.)

MEASURING START.

  10, 63(0):0.40, 63(0):0.42, 7(0):0.76, 6(0):2.60, 6(0):0.17
  20, 75(0):0.40, 74(0):0.62, 7(0):0.04, 9(0):2.38, 7(0):0.75
  30, 83(0):0.22, 84(0):0.37, 9(0):0.04, 7(0):1.97, 9(0):0.80

STOPPING THREADS.....

<RT Histogram>

1.New-Order
2.Payment
3.Order-Status
4.Delivery
5.Stock-Level

<90th Percentile RT (MaxRT)>
   New-Order : 0.37  (1.10)
     Payment : 0.47  (1.24)
Order-Status : 0.06  (0.96)
    Delivery : 2.43  (2.72)
 Stock-Level : 0.75  (0.79)

<Raw Results>
  [0] sc:221  lt:0  rt:0  fl:0
  [1] sc:221  lt:0  rt:0  fl:0
  [2] sc:23  lt:0  rt:0  fl:0
```

```
  [3] sc:22  lt:0  rt:0  fl:0
  [4] sc:22  lt:0  rt:0  fl:0
 in 30 sec.

<Raw Results2(sum ver.)>
  [0] sc:221  lt:0  rt:0  fl:0
  [1] sc:221  lt:0  rt:0  fl:0
  [2] sc:23  lt:0  rt:0  fl:0
  [3] sc:22  lt:0  rt:0  fl:0
  [4] sc:22  lt:0  rt:0  fl:0

<Constraint Check> (all must be [OK])
 [transaction percentage]
        Payment: 43.42% (>=43.0%) [OK]
   Order-Status: 4.52% (>= 4.0%) [OK]
       Delivery: 4.32% (>= 4.0%) [OK]
    Stock-Level: 4.32% (>= 4.0%) [OK]
 [response time (at least 90% passed)]
      New-Order: 100.00%  [OK]
        Payment: 100.00%  [OK]
   Order-Status: 100.00%  [OK]
       Delivery: 100.00%  [OK]
    Stock-Level: 100.00%  [OK]

<TpmC>
                 442.000 TpmC
```

最后一行就是测试的结果：每分钟执行完的事务数[注11]。如果紧挨着最后一行前发现有异常结果输出，比如有关于约束检查的信息，那么可以检查一下响应时间的直方图，或者通过其他详细输出信息寻找线索。当然，最好是能使用本章前面提到的一些脚本，这样就可以很容易获得测试执行期间的详细的诊断数据和性能数据。

66

2.6 总结

每个 MySQL 的使用者都应该了解一些基准测试的知识。基准测试不仅仅是用来解决业务问题的一种实践行动，也是一种很好的学习方法。学习如何将问题分解成可以通过基准测试来获得答案的方法，就和在数学课上从文字题目中推导出方程式一样。首先正确地描述问题，之后选择合适的基准测试来回答问题，设置基准测试的持续时间和参数，运行测试，收集数据，分析结果数据，这一系列的训练可以帮助你成为更好的 MySQL 用户。

如果你还没有做过基准测试，那么建议至少要熟悉 *sysbench*。可以先学习如何使用 `oltp` 和 `fileio` 测试。`oltp` 基准测试可以很方便地比较不同系统的性能。另一方面，文件系统和磁盘基准测试，则可以在系统出现问题时有效地诊断和隔离异常的组件。通过这样的 67

注 11： 我们是在笔记本电脑上运行这个基准测试的，这只是作为演示用的。真实服务器的速度肯定比这快得多。

基准测试，我们多次发现了一些数据库管理员的说法存在问题，比如SAN存储真的出现了一块坏盘，或者RAID控制器的缓存策略的配置并不是像工具中显示的那样。通过对单块磁盘进行基准测试，如果发现每秒可以执行14 000次随机读，那要么是碰到了严重的错误，要么是配置出现了问题[注12]。

如果经常执行基准测试，那么制定一些原则是很有必要的。选择一些合适的测试工具并深入地学习。可以建立一个脚本库，用于配置基准测试，收集输出结果、系统性能和状态信息，以及分析结果。使用一种熟练的绘图工具如 *gnuplot* 或者 *R*（不用浪费时间使用电子表格，它们既笨重，速度又慢）。尽量早和多地使用绘图的方式，来发现基准测试和系统中的问题和错误。你的眼睛是比任何脚本和自动化工具都更有效的发现问题的工具。

注12：一块机械磁盘每秒只能执行几百次的随机读操作，因为寻道操作是需要时间的。

第 3 章 69

服务器性能剖析

在我们的技术咨询生涯中，最常碰到的三个性能相关的服务请求是：如何确认服务器是否达到了性能最佳的状态、找出某条语句为什么执行不够快，以及诊断被用户描述成“停顿”、“堆积”或者“卡死”的某些间歇性疑难故障。本章将主要针对这三个问题做出解答。我们将提供一些工具和技巧来优化整机的性能、优化单条语句的执行速度，以及诊断或者解决那些很难观察到的问题（这些问题用户往往很难知道其根源，有时候甚至都很难察觉到它的存在）。

这看起来是个艰巨的任务，但是事实证明，有一个简单的方法能够从噪声中发现苗头。这个方法就是专注于测量服务器的时间花费在哪里，使用的技术则是性能剖析（profiling）。在本章，我们将展示如何测量系统并生成剖析报告，以及如何分析系统的整个堆栈（stack），包括从应用程序到数据库服务器到单个查询。

首先我们要保持空杯精神，抛弃掉一些关于性能的常见的误解。这有一定的难度，下面我们一起通过一些例子来说明问题在哪里。

3.1 性能优化简介

问 10 个人关于性能的问题，可能会得到 10 个不同的回答，比如“每秒查询次数”、“CPU 利用率”、“可扩展性”之类。这其实也没有问题，每个人在不同场景下对性能有不同的理解，但本章将给性能一个正式的定义。我们将性能定义为完成某件任务所需要的时间度量，换句话说，性能即响应时间，这是一个非常重要的原则。我们通过任务和时间而不是资源来测量性能。数据库服务器的目的是执行 SQL 语句，所以它关注的任务是查询或者语句，如 SELECT、UPDATE、DELETE 等[注1]。数据库服务器的性能用查询的响应时间来 70

注 1： 本书不会严格区分查询和语句，DDL 和 DML 等。不管给服务器发送什么命令，关心的都是执行命令的速度。本书将使用“查询”一词泛指所有发送给服务器的命令。

度量，单位是每个查询花费的时间。

还有另外一个问题：什么是优化？我们暂时不讨论这个问题，而是假设性能优化就是在一定的工作负载下尽可能地[注2]降低响应时间。

很多人对此很迷茫。假如你认为性能优化是降低 CPU 利用率，那么可以减少对资源的使用。但这是一个陷阱，资源是用来消耗并用来工作的，所以有时候消耗更多的资源能够加快查询速度。很多时候将使用老版本 InnoDB 引擎的 MySQL 升级到新版本后，CPU 利用率会上升得很厉害，这并不代表性能出现了问题，反而说明新版本的 InnoDB 对资源的利用率上升了。查询的响应时间则更能体现升级后的性能是不是变得更好。版本升级有时候会带来一些 bug，比如不能利用某些索引从而导致 CPU 利用率上升。CPU 利用率只是一种现象，而不是很好的可度量的目标。

同样，如果把性能优化仅仅看成是提升每秒查询量，这其实只是吞吐量优化。吞吐量的提升可以看作性能优化的副产品[注3]。对查询的优化可以让服务器每秒执行更多的查询，因为每条查询执行的时间更短了（吞吐量的定义是单位时间内的查询数量，这正好是我们对性能的定义的倒数）。

所以如果目标是降低响应时间，那么就需要理解为什么服务器执行查询需要这么多时间，然后去减少或者消除那些对获得查询结果来说不必要的工作。也就是说，先要搞清楚时间花在哪里。这就引申出优化的第二个原则：无法测量就无法有效地优化。所以第一步应该测量时间花在什么地方。

71 我们观察到，很多人在优化时，都将精力放在修改一些东西上，却很少去进行精确的测量。我们的做法完全相反，将花费非常多，甚至 90% 的时间来测量响应时间花在哪里。如果通过测量没有找到答案，那要么是测量的方式错了，要么是测量得不够完整。如果测量了系统中完整而且正确的数据，性能问题一般都能暴露出来，对症下药的解决方案也就比较明了。测量是一项很有挑战性的工作，并且分析结果也同样有挑战性，测出时间花在哪里，和知道为什么花在那里，是两码事。

前面提到需要合适的测量范围，这是什么意思呢？合适的测量范围是说只测量需要优化的活动。有两种比较常见的情况会导致不合适的测量：

注 2： 本书尽量避免从理论上来阐述性能优化一词，如果有兴趣可以参考阅读另外两篇文章。在 Percona 的网站（*http://www.percona.com*）上，有一篇名为 *Goal-Driven Performance Optimization* 的白皮书，这是一篇紧凑的快速参考页。另外一篇是 Cary Millsap 的 *Optimizing Oracle Performance*（O'Reilly 出版）。Cary 的优化方法，被称为 R 方法，是 Oracle 世界的优化黄金定律。

注 3： 也有人将优化定义为提升吞吐量，这也没有什么问题，但本书采用的不是这个定义，因为我们认为响应时间更重要，尽管吞吐量在基准测试中更容易测量。

- 在错误的时间启动和停止测量。
- 测量的是聚合后的信息，而不是目标活动本身。

例如，一个常见的错误是先查看慢查询，然后又去排查整个服务器的情况来判断问题在哪里。如果确认有慢查询，那么就应该测量慢查询，而不是测量整个服务器。测量的应该是从慢查询的开始到结束的时间，而不是查询之前或查询之后的时间。

完成一项任务所需要的时间可以分成两部分：执行时间和等待时间。如果要优化任务的执行时间，最好的办法是通过测量定位不同的子任务花费的时间，然后优化去掉一些子任务、降低子任务的执行频率或者提升子任务的效率。而优化任务的等待时间则相对要复杂一些，因为等待有可能是由其他系统间接影响导致，任务之间也可能由于争用磁盘或者 CPU 资源而相互影响。根据时间是花在执行还是等待上的不同，诊断也需要不同的工具和技术。

刚才说到需要定位和优化子任务，但只是一笔带过。一些运行不频繁或者很短的子任务对整体响应时间的影响很小，通常可以忽略不计。那么如何确认哪些子任务是优化的目标呢？这个时候性能剖析就可以派上用场了。

如何判断测量是正确的？

72

如果测量是如此重要，那么测量错了会有什么后果？实际上，测量经常都是错误的。对数量的测量并不等于数量本身。测量的错误可能很小，跟实际情况区别不大，但错的终归是错的。所以这个问题其实应该是："测量到底有多么不准确？"这个问题在其他一些书中有详细的讨论，但不是本书的主题。但是要意识到使用的是测量数据，而不是其所代表的实际数据。通常来说，测量的结果也可能有多种模糊的表现，这可能导致推断出错误的结论。

3.1.1 通过性能剖析进行优化

一旦掌握并实践面向响应时间的优化方法，就会发现需要不断地对系统进行性能剖析（profiling）。

性能剖析是测量和分析时间花费在哪里的主要方法。性能剖析一般有两个步骤：测量任务所花费的时间；然后对结果进行统计和排序，将重要的任务排到前面。

性能剖析工具的工作方式基本相同。在任务开始时启动计时器，在任务结束时停止计时器，然后用结束时间减去启动时间得到响应时间。也有些工具会记录任务的父任务。这些结果数据可以用来绘制调用关系图，但对于我们的目标来说更重要的是，可以将相似的任务分组并进行汇总。对相似的任务分组并进行汇总可以帮助对那些分到一组的任务做更复杂的统计分析，但至少需要知道每一组有多少任务，并计算出总的响应时间。通过性能剖析报告（*profile* report）可以获得需要的结果。性能剖析报告会列出所有任务列表。每行记录一个任务，包括任务名、任务的执行时间、任务的消耗时间、任务的平均执行时间，以及该任务执行时间占全部时间的百分比。性能剖析报告会按照任务的消耗时间进行降序排序。

为了更好地说明，这里举一个对整个数据库服务器工作负载的性能剖析的例子，主要输出的是各种类型的查询和执行查询的时间。这是从整体的角度来分析响应时间，后面会演示其他角度的分析结果。下面的输出是用 Percona Toolkit 中的 *pt-query-digest*（实际上就是著名的 Maatkit 工具中的 *mk-query-digest*）分析得到的结果。为了显示方便，对结果做了一些微调，并且只截取了前面几行结果：

```
Rank Response time    Calls R/Call Item
==== ================ ===== ====== =======
   1 11256.3618 68.1% 78069 0.1442 SELECT InvitesNew
   2  2029.4730 12.3% 14415 0.1408 SELECT StatusUpdate
   3  1345.3445  8.1%  3520 0.3822 SHOW STATUS
```

73 上面只是性能剖析结果的前几行，根据总响应时间进行排名，只包括剖析所需要的最小列组合。每一行都包括了查询的响应时间和占总时间的百分比、查询的执行次数、单次执行的平均响应时间，以及该查询的摘要。通过这个性能剖析可以很清楚地看到每个查询相互之间的成本比较，以及每个查询占总成本的比较。在这个例子中，任务指的就是查询，实际上在分析 MySQL 的时候经常都指的是查询。

我们将实际地讨论两种类型的性能剖析：基于执行时间的分析和基于等待的分析。基于执行时间的分析研究的是什么任务的执行时间最长，而基于等待的分析则是判断任务在什么地方被阻塞的时间最长。

如果任务执行时间长是因为消耗了太多的资源且大部分时间花费在执行上，等待的时间不多，这种情况下基于等待的分析作用就不大。反之亦然，如果任务一直在等待，没有消耗什么资源，去分析执行时间就不会有什么结果。如果不能确认问题是出在执行还是等待上，那么两种方式都需要试试。后面会给出详细的例子。

事实上，当基于执行时间的分析发现一个任务需要花费太多时间的时候，应该深入去分析一下，可能会发现某些“执行时间”实际上是在等待。例如，上面简单的性能剖析的输出显示表 InvitesNew 上的 SELECT 查询花费了大量时间，如果深入研究，则可能发现

时间都花费在等待 I/O 完成上。

在对系统进行性能剖析前，必须先要能够进行测量，这需要系统可测量化的支持。可测量的系统一般会有多个测量点可以捕获并收集数据，但实际系统很少可以做到可测量化。大部分系统都没有多少可测量点，即使有也只提供一些活动的计数，而没有活动花费的时间统计。MySQL 就是一个典型的例子，直到版本 5.5 才第一次提供了 Performance Schema，其中有一些基于时间的测量点[注4]，而版本 5.1 及之前的版本没有任何基于时间的测量点。能够从 MySQL 收集到的服务器操作的数据大多是 `show status` 计数器的形式，这些计数器统计的是某种活动发生的次数。这也是我们最终决定创建 Percona Server 的主要原因，Percona Server 从版本 5.0 开始提供很多更详细的查询级别的测量点。

虽然理想的性能优化技术依赖于更多的测量点，但幸运的是，即使系统没有提供测量点，也还有其他办法可以展开优化工作。因为还可以从外部去测量系统，如果测量失败，也可以根据对系统的了解做出一些靠谱的猜测。但这么做的时候一定要记住，不管是外部 74
测量还是猜测，数据都不是百分之百准确的，这是系统不透明所带来的风险。

举个例子，在 Percona Server 5.0 中，慢查询日志揭露了一些性能低下的原因，如磁盘 I/O 等待或者行级锁等待。如果日志中显示一条查询花费 10 秒，其中 9.6 秒在等待磁盘 I/O，那么追究其他 4% 的时间花费在哪里就没有意义，磁盘 I/O 才是最重要的原因。

3.1.2 理解性能剖析

MySQL 的性能剖析（profile）将最重要的任务展示在前面，但有时候没显示出来的信息也很重要。可以参考一下前面提到过的性能剖析的例子。不幸的是，尽管性能剖析输出了排名、总计和平均值，但还是有很多需要的信息是缺失的，如下所示。

值得优化的查询（worthwhile query）

　　性能剖析不会自动给出哪些查询值得花时间去优化。这把我们带回到优化的本意，如果你读过 Cary Millsap 的书，对此就会有更多的理解。这里我们要再次强调两点：第一，一些只占总响应时间比重很小的查询是不值得优化的。根据阿姆达尔定律（Amdahl's Law），对一个占总响应时间不超过 5% 的查询进行优化，无论如何努力，收益也不会超过 5%。第二，如果花费了 1 000 美元去优化一个任务，但业务的收入没有任何增加，那么可以说反而导致业务被逆优化了 1 000 美元。如果优化的成本大于收益，就应当停止优化。

异常情况

　　某些任务即使没有出现在性能剖析输出的前面也需要优化。比如某些任务执行次数很少，但每次执行都非常慢，严重影响用户体验。因为其执行频率低，所以总的响

注 4：　MySQL 5.5 的 Performance Schema 也没有提供查询级别的细节数据，要到 MySQL 5.6 才提供。

应时间占比并不突出。

未知的未知[注5]

一款好的性能剖析工具会显示可能的“丢失的时间”。丢失的时间指的是任务的总时间和实际测量到的时间之间的差。例如，如果处理器的 CPU 时间是 10 秒，而剖析到的任务总时间是 9.7 秒，那么就有 300 毫秒的丢失时间。这可能是有些任务没有测量到，也可能是由于测量的误差和精度问题的缘故。如果工具发现了这类问题，则要引起重视，因为有可能错过了某些重要的事情。即使性能剖析没有发现丢失时间，
75 也需要注意考虑这类问题存在的可能性，这样才不会错过重要的信息。我们的例子中没有显示丢失的时间，这是我们所使用工具的一个局限性。

被掩藏的细节

性能剖析无法显示所有响应时间的分布。只相信平均值是非常危险的，它会隐藏很多信息，而且无法表达全部情况。Peter 经常举例说医院所有病人的平均体温没有任何价值[注6]。假如在前面的性能剖析的例子的第一项中，如果有两次查询的响应时间是 1 秒，而另外 12 771 次查询的响应时间是几十微秒，结果会怎样？只从平均值里是无法发现两次 1 秒的查询的。要做出最好的决策，需要为性能剖析里输出的这一行中包含的 12 773 次查询提供更多的信息，尤其是更多响应时间的信息，比如直方图、百分比、标准差、偏差指数等。

好的工具可以自动地获得这些信息。实际上，*pt-query-digest* 就在剖析的结果里包含了很多这类细节信息，并且输出在剖析报告中。对此我们做了简化，可以将精力集中在重要而基础的例子上：通过排序将最昂贵的任务排在前面。本章后面会展示更多丰富而有用的性能剖析的例子。

在前面的性能剖析的例子中，还有一个重要的缺失，就是无法在更高层次的堆栈中进行交互式的分析。当我们仅仅着眼于服务器中的单个查询时，无法将相关查询联系起来，也无法理解这些查询是否是同一个用户交互的一部分。性能剖析只能管中窥豹，而无法将剖析从任务扩展至事务或者页面查看（page view）的级别。也有一些办法可以解决这个问题，比如给查询加上特殊的注释作为标签，可以标明其来源并据此做聚合，也可以在应用层面增加更多的测量点，这是下一节的主题。

3.2 对应用程序进行性能剖析

对任何需要消耗时间的任务都可以做性能剖析，当然也包括应用程序。实际上，剖析应用程序一般比剖析数据库服务器容易，而且回报更多。虽然前面的演示例子都是针对

注 5：　在此向 Donald Rumsfeld 道歉。他的评论尽管听起来可笑，但实际上非常有见地。
注 6：　啊！（这只是个玩笑，我们并不坚持。）

MySQL 服务器的剖析，但对系统进行性能剖析还是建议自上而下地进行[注7]，这样可以追踪自用户发起到服务器响应的整个流程。虽然性能问题大多数情况下都和数据库有关，但应用导致的性能问题也不少。性能瓶颈可能有很多影响因素： 76

- 外部资源，比如调用了外部的 Web 服务或者搜索引擎。
- 应用需要处理大量的数据，比如分析一个超大的 XML 文件。
- 在循环中执行昂贵的操作，比如滥用正则表达式。
- 使用了低效的算法，比如使用暴力搜索算法（naïve search algorithm）来查找列表中的项。

幸运的是，确定 MySQL 的问题没有这么复杂，只需要一款应用程序的剖析工具即可（作为回报，一旦拥有这样的工具，就可以从一开始就写出高效的代码）。

建议在所有的新项目中都考虑包含性能剖析的代码。往已有的项目中加入性能剖析代码也许很困难，新项目就简单一些。

性能剖析本身会导致服务器变慢吗？

说“是的”，是因为性能剖析确实会导致应用慢一点；说“不是”，是因为性能剖析可以帮助应用运行得更快。先别急，下面就解释一下为什么这么说。

性能剖析和定期检测都会带来额外开销。问题在于这部分的开销有多少，并且由此获得的收益是否能够抵消这些开销。

大多数设计和构建过高性能应用程序的人相信，应该尽可能地测量一切可以测量的地方，并且接受这些测量带来的额外开销，这些开销应该被当成应用程序的一部分。Oracle 的性能优化大师 Tom Kyte 曾被问到 Oracle 中的测量点的开销，他的回答是，测量点至少为性能优化贡献了 10%。对此我们深表赞同，而且大多数应用并不需要每天都运行详细的性能测量，所以实际贡献甚至要超过 10%。即使不同意这个观点，为应用构建一些可以永久使用的轻量级的性能剖析也是有意义的。如果系统没有每天变化的性能统计，则碰到无法提前预知的性能瓶颈就是一件头痛的事情。发现问题的时候，如果有历史数据，则这些历史数据价值是无限的。而且性能数据还可以帮助规划好硬件采购、资源分配，以及预测周期性的性能尖峰。

注 7：我们将在后面展示例子，因为需要有一些先验知识，这个问题跟底层相关，所以我们先跳过自顶向下的方法。

那么何谓“轻量级”的性能剖析？比如可以为所有 SQL 语句计时，加上脚本总时间统计，这样做的代价不高，而且不需要在每次页面查看（page view）时都执行。如果流量趋势比较稳定，随机采样也可以，随机采样可以通过在应用程序中设置实现：

77

```
<?php
$profiling_enabled = rand(0, 100) > 99;
?>
```

这样只有 1% 的会话会执行性能采样，来帮助定位一些严重的问题。这种策略在生产环境中尤其有用，可以发现一些其他方法无法发现的问题。

几年前在写作本书的第二版的时候，流行的 Web 编程语言和框架中还没有太多现成的性能剖析工具可以用于生产环境，所以在书中展示了一段示例代码，可以简单而有效地复制使用。而到了今天，已经有了很多好用的工具，要做的只是打开工具箱，就可以开始优化性能。

首先，这里要“兜售”的一个好工具是一款叫做 New Relic 的软件即服务（software-as-a-service）产品。声明一下我们不是“托”，我们一般不会推荐某个特定公司或产品，但这个工具真的非常棒，建议大家都用它。我们的客户借助这个工具，在没有我们帮助的情况下，解决了很多问题；即使有时候找不到解决办法，但依然能够帮助定位到问题。New Relic 会插入到应用程序中进行性能剖析，将收集到的数据发送到一个基于 Web 的仪表盘，使用仪表盘可以更容易利用面向响应时间的方法分析应用性能。这样用户只需要考虑做那些正确的事情，而不用考虑如何去做。而且 New Relic 测量了很多用户体验相关的点，涵盖从 Web 浏览器到应用代码，再到数据库及其他外部调用。

像 New Relic 这类工具的好处是可以全天候地测量生产环境的代码——既不限于测试环境，也不限于某个时间段。这一点非常重要，因为有很多剖析工具或者测量点的代价很高，所以不能在生产环境全天候运行。在生产环境运行，可以发现一些在测试环境和预发环境无法发现的性能问题。如果工具在生产环境全天候运行的成本太高，那么至少也要在集群中找一台服务器运行，或者只针对部分代码运行，原因请参考前面的“性能剖析本身会导致服务器变慢吗？”。

3.2.1 测量 PHP 应用程序

如果不使用 New Relic，也有其他的选择。尤其是对 PHP，有好几款工具都可以帮助进行性能剖析。其中一款叫做 *xhprof*（*http://pecl.php.net/package/xhprof*），这是 Facebook

开发给内部使用的，在 2009 年开源了。*xhprof* 有很多高级特性，并且易于安装和使用，它很轻量级，可扩展性也很好，可以在生产环境大量部署并全天候使用，它还能针对函数调用进行剖析，并根据耗费的时间进行排序。相比 *xhprof*，还有一些更底层的工具，比如 *xdebug*、*Valgrind* 和 *cachegrind*，可以从多个角度对代码进行检测[注8]。有些工具会产生大量输出，并且开销很大，并不适合在生产环境运行，但在开发环境却可以发挥很大的作用。 78

下面要讨论的另外一个 PHP 性能剖析工具是我们自己写的，基于本书第二版的代码和原则扩展而来，名叫 IfP（instrumentation-for-php），代码托管在 Goole Code 上（*http://code.google.com/p/instrumentation-for-php/*）。Ifp 并不像 *xhprof* 一样对 PHP 做深入的测量，而是更关注数据库调用。所以当无法在数据库层面进行测量的时候，Ifp 可以很好地帮助应用剖析数据库的利用率。Ifp 是一个提供了计数器和计时器的单例类，很容易部署到生产环境中，因为不需要访问 PHP 配置的权限（对很多开发人员来说，都没有访问 PHP 配置的权限，所以这一点很重要）。

Ifp 不会自动剖析所有的 PHP 函数，而只是针对重要的函数。例如，对于某些需要剖析的地方要用到自定义的计数器，就需要手工启动和停止。但 Ifp 可以自动对整个页面的执行进行计时，这样对自动测量数据库和 *memcached* 的调用就比较简单，对于这种情况就无须手工启动或者停止。这也意味着，Ifp 可以剖析三种情况：应用程序的请求（如 page view）、数据库的查询和缓存的查询。Ifp 还可以将计数器和计时器输出到 Apache，通过 Apache 可以将结果写入到日志中。这是一种方便且轻量的记录结果的方式。Ifp 不会保存其他数据，所以也不需要有系统管理员的权限。

使用 Ifp，只需要简单地在页面的开始处调用 `start_request()`。理想情况下，在程序的一开始就应当调用：

```
require_once('Instrumentation.php');
Instrumentation::get_instance()->start_request();
```

这段代码注册了一个 shutdown 函数，所以在执行结束的地方不需要再做更多的处理。

Ifp 会自动对 SQL 添加注释，便于从数据库的查询日志中更灵活地分析应用的情况，通过 `SHOW PROCESSLIST` 也可以更清楚地知道性能低的查询出自何处。大多数情况下，定位性能低下查询的来源都不容易，尤其是那些通过字符串拼接出来的查询语句，都没有办法在源代码中去搜索。那么 Ifp 的这个功能就可以帮助解决这个问题，它可以很快定位到查询是从何处而来的，即使应用和数据库中间加了代理或者负载均衡层，也可以确认是哪个应用的用户，是哪个页面请求，是源代码中的哪个函数、代码行号，甚至是所创 79

注 8： 不像 PHP，大部分其他编程语言都有一些内建的剖析功能。例如 Ruby 可以使用 *-r* 选项，Perl 则可以使用 *perl -d:DProf*，等等。

建的计数器的键值对。下面是一个例子：

```
-- File: index.php Line: 118 Function: fullCachePage request_id: ABC session_id: XYZ
SELECT * FROM ...
```

如何测量 MySQL 的调用取决于连接 MySQL 的接口。如果使用的是面向对象的 *mysqli* 接口，则只需要修改一行代码：将构造函数从 *mysqli* 改为可以自动测量的 *mysqli_x* 即可。*mysqli_x* 构造函数是由 Ifp 提供的子类，可以在后台测量并改写查询。如果使用的不是面向对象的接口，或者是其他的数据库访问层，则需要修改更多的代码。如果数据库调用不是分散在代码各处还好，否则建议使用集成开发环境（IDE）如 Eclipse，这样修改起来要容易些。但不管从哪个方面来看，将访问数据库的代码集中到一起都可以说是最佳实践。

Ifp 的结果很容易分析。Percona Toolkit 中的 *pt-query-digest* 能够很方便地从查询注释中抽取出键值对，所以只需要简单地将查询记录到 MySQL 的日志文件中，再对日志文件进行处理即可。Apache 的 *mod_log_config* 模块可以利用 Ifp 输出的环境变量来定制日志输出，其中的宏 `%D` 还可以以微秒级记录请求时间。

也可以通过 `LOAD DATA INFILE` 将 Apache 的日志载入到 MySQL 数据库中，然后通过 SQL 进行查询。在 Ifp 的网站上有一个 PDF 的幻灯片，详细给出了使用示例，包括查询和命令行参数都有。

或许你会说不想或者没时间在代码中加入测量的功能，其实这事比想象的要容易得多，而且花在优化上的时间将会由于性能的优化而加倍地回报给你。对应用的测量是不可替代的。当然最好是直接使用 New Relic、*xhprof*、Ifp 或者其他已有的优化工具，而不必重新去发明“轮子”。

80

MySQL 企业监控器的查询分析功能

MySQL 的企业监控器（Enterprise Monitor）也是值得考虑的工具之一。这是 Oracle 提供的 MySQL 商业服务支持中的一部分。它可以捕获发送给服务器的查询，要么是通过应用程序连接 MySQL 的库文件实现，要么是在代理层实现（我们并不太建议使用代理层）。该工具有设计良好的用户界面，可以直观地显示查询的剖析结果，并且可以根据时间段进行缩放，例如可以选择某个异常的性能尖峰时间来查看状态图。也可以查看 `EXPLAIN` 出来的执行计划，这在故障诊断时非常有用。

3.3 剖析 MySQL 查询

对查询进行性能剖析有两种方式，每种方式都有各自的问题，本章会详细介绍。可以剖析整个数据库服务器，这样可以分析出哪些查询是主要的压力来源（如果已经在最上面的应用层做过剖析，则可能已经知道哪些查询需要特别留意）。定位到具体需要优化的查询后，也可以钻取下去对这些查询进行单独的剖析，分析哪些子任务是响应时间的主要消耗者。

3.3.1 剖析服务器负载

服务器端的剖析很有价值，因为在服务器端可以有效地审计效率低下的查询。定位和优化“坏”查询能够显著地提升应用的性能，也能解决某些特定的难题。还可以降低服务器的整体压力，这样所有的查询都将因为减少了对共享资源的争用而受益（“间接的好处”）。降低服务器的负载也可以推迟或者避免升级更昂贵硬件的需求，还可以发现和定位糟糕的用户体验，比如某些极端情况。

MySQL 的每一个新版本中都增加了更多的可测量点。如果当前的趋势可靠的话，那么在性能方面比较重要的测量需求很快能够在全球范围内得到支持。但如果只是需要剖析并找出代价高的查询，就不需要如此复杂。有一个工具很早之前就能帮到我们了，这就是慢查询日志。

捕获 MySQL 的查询到日志文件中

在 MySQL 中，慢查询日志最初只是捕获比较“慢”的查询，而性能剖析却需要针对所有的查询。而且在 MySQL 5.0 及之前的版本中，慢查询日志的响应时间的单位是秒，粒度太粗了。幸运的是，这些限制都已经成为历史了。在 MySQL 5.1 及更新的版本中，慢日志的功能已经被加强，可以通过设置 `long_query_time` 为 0 来捕获所有的查询，而且 81
查询的响应时间单位已经可以做到微秒级。如果使用的是 Percona Server，那么 5.0 版本就具备了这些特性，而且 Percona Server 提供了对日志内容和查询捕获的更多控制能力。

在 MySQL 的当前版本中，慢查询日志是开销最低、精度最高的测量查询时间的工具。如果还在担心开启慢查询日志会带来额外的 I/O 开销，那大可以放心。我们在 I/O 密集型场景做过基准测试，慢查询日志带来的开销可以忽略不计（实际上在 CPU 密集型场景的影响还稍微大一些）。更需要担心的是日志可能消耗大量的磁盘空间。如果长期开启慢查询日志，注意要部署日志轮转（log rotation）工具。或者不要长期启用慢查询日志，只在需要收集负载样本的期间开启即可。

MySQL 还有另外一种查询日志，被称之为“通用日志”，但很少用于分析和剖析服务器性能。通用日志在查询请求到服务器时进行记录，所以不包含响应时间和执行计划等重

要信息。MySQL 5.1 之后支持将日志记录到数据库的表中，但多数情况下这样做没什么必要。这不但对性能有较大影响，而且 MySQL 5.1 在将慢查询记录到文件中时已经支持微秒级别的信息，然而将慢查询记录到表中会导致时间粒度退化为只能到秒级。而秒级别的慢查询日志没有太大的意义。

Percona Server 的慢查询日志比 MySQL 官方版本记录了更多细节且有价值的信息，如查询执行计划、锁、I/O 活动等。这些特性都是随着处理各种不同的优化场景的需求而慢慢加进来的。另外在可管理性上也进行了增强。比如全局修改针对每个连接的 *long_query_time* 的阈值，这样当应用使用连接池或者持久连接的时候，可以不用重置会话级别的变量而启动或者停止连接的查询日志。总的来说，慢查询日志是一种轻量而且功能全面的性能剖析工具，是优化服务器查询的利器。

有时因为某些原因如权限不足等，无法在服务器上记录查询。这样的限制我们也常常碰到，所以我们开发了两种替代的技术，都集成到了 Percona Toolkit 中的 *pt-query-digest* 中。第一种是通过 *--processlist* 选项不断查看 `SHOW FULL PROCESSLIST` 的输出，记录查询第一次出现的时间和消失的时间。某些情况下这样的精度也足够发现问题，但却无法捕获所有的查询。一些执行较快的查询可能在两次执行的间隙就执行完成了，从而无法捕获到。

第二种技术是通过抓取 TCP 网络包，然后根据 MySQL 的客户端 / 服务端通信协议进行解析。可以先通过 *tcpdump* 将网络包数据保存到磁盘，然后使用 *pt-query-digest* 的 *--type=tcpdump* 选项来解析并分析查询。此方法的精度比较高，并且可以捕获所有查询。还可以解析更高级的协议特性，比如可以解析二进制协议，从而创建并执行服务端
82 预解析的语句（prepared statement）及压缩协议。另外还有一种方法，就是通过 MySQL Proxy 代理层的脚本来记录所有查询，但在实践中我们很少这样做。

分析查询日志

强烈建议大家从现在起就利用慢查询日志捕获服务器上的所有查询，并且进行分析。可以在一些典型的时间窗口如业务高峰期的一个小时内记录查询。如果业务趋势比较均衡，那么一分钟甚至更短的时间内捕获需要优化的低效查询也是可行的。

不要直接打开整个慢查询日志进行分析，这样做只会浪费时间和金钱。首先应该生成一个剖析报告，如果需要，则可以再查看日志中需要特别关注的部分。自顶向下是比较好的方式，否则有可能像前面提到的，反而导致业务的逆优化。

从慢查询日志中生成剖析报告需要有一款好工具，这里我们建议使用 *pt-query-digest*，这毫无疑问是分析 MySQL 查询日志最有力的工具。该工具功能强大，包括可以将查询

报告保存到数据库中，以及追踪工作负载随时间的变化。

一般情况下，只需要将慢查询日志文件作为参数传递给 *pt-query-digest*，就可以正确地工作了。它会将查询的剖析报告打印出来，并且能够选择将“重要”的查询逐条打印出更详细的信息。输出的报告细节详尽，绝对可以让生活更美好。该工具还在持续的开发中，因此要了解最新的功能请阅读最新版本的文档。

这里给出一份 *pt-query-digest* 输出的报告的例子，作为进行性能剖析的开始。这是前面提到过的一个未修改过的剖析报告：

```
# Profile
# Rank Query ID           Response time    Calls R/Call V/M   Item
# ==== ================== ================ ===== ====== ===== =======
#    1 0xBFCF8E3F293F6466 11256.3618 68.1% 78069 0.1442  0.21 SELECT InvitesNew?
#    2 0x620B8CAB2B1C76EC  2029.4730 12.3% 14415 0.1408  0.21 SELECT StatusUpdate?
#    3 0xB90978440CC11CC7  1345.3445  8.1%  3520 0.3822  0.00 SHOW STATUS
#    4 0xCB73D6B5B031B4CF  1341.6432  8.1%  3509 0.3823  0.00 SHOW STATUS
# MISC 0xMISC               560.7556  3.4% 23930 0.0234   0.0 <17 ITEMS>
```

可以看到这个比之前的版本多了一些细节。首先，每个查询都有一个 ID，这是对查询语句计算出的哈希值指纹，计算时去掉了查询条件中的文本值和所有空格，并且全部转化为小写字母（请注意第三条和第四条语句的摘要看起来一样，但哈希指纹是不一样的）。该工具对表名也有类似的规范做法。表名 `InvitesNew` 后面的问号意味着这是一个分片（shard）的表，表名后面的分片标识被问号替代，这样就可以将同一组分片表作为一个整体做汇总统计。这个例子实际上是来自一个压力很大的分片过的 Facebook 应用。

报告中的 V/M 列提供了方差均值比（variance-to-mean ratio）的详细数据，方差均值比 83
也就是常说的离差指数（index of dispersion）。离差指数高的查询对应的执行时间的变化较大，而这类查询通常都值得去优化。如果 *pt-query-digest* 指定了 *--explain* 选项，输出结果中会增加一列简要描述查询的执行计划，执行计划是查询背后的“极客代码”。通过联合观察执行计划列和 V/M 列，可以更容易识别出性能低下需要优化的查询。

最后，在尾部也增加了一行输出，显示了其他 17 个占比较低而不值得单独显示的查询的统计数据。可以通过 *--limit* 和 *--outliers* 选项指定工具显示更多查询的详细信息，而不是将一些不重要的查询汇总在最后一行。默认只会打印时间消耗前 10 位的查询，或者执行时间超过 1 秒阈值很多倍的查询，这两个限制都是可配置的。

剖析报告的后面包含了每种查询的详细报告。可以通过查询的 ID 或者排名来匹配前面的剖析统计和查询的详细报告。下面是排名第一也就是“最差”的查询的详细报告：

```
# Query 1: 24.28 QPS, 3.50x concurrency, ID 0xBFCF8E3F293F6466 at byte 5590079
# This item is included in the report because it matches --limit.
# Scores: V/M = 0.21
# Query_time sparkline: | _^_.^_ |
# Time range: 2008-09-13 21:51:55 to 22:45:30
# Attribute    pct   total     min     max     avg     95%  stddev  median
# ============ === ======= ======= ======= ======= ======= ======= =======
# Count         63   78069
# Exec time     68  11256s    37us      1s   144ms   501ms   175ms    68ms
# Lock time     85    134s       0   650ms     2ms   176us    20ms    57us
# Rows sent      8  70.18k       0       1    0.92    0.99    0.27    0.99
# Rows examine   8  70.84k       0       3    0.93    0.99    0.28    0.99
# Query size    84  10.43M     135     141  140.13  136.99    0.10  136.99
# String:
# Databases     production
# Hosts
# Users         fbappuser
# Query_time distribution
#   1us
#  10us  #
# 100us  ####################################################
#   1ms  ###
#  10ms  ################
# 100ms  ################################################################
#    1s  #
#  10s+
# Tables
#    SHOW TABLE STATUS FROM `production ` LIKE'InvitesNew82'\G
#    SHOW CREATE TABLE `production `.`InvitesNew82'\G
# EXPLAIN /*!50100 PARTITIONS*/
SELECT InviteId, InviterIdentifier FROM InvitesNew82 WHERE (InviteSetId = 87041469)
AND (InviteeIdentifier = 1138714082) LIMIT 1\G
```

84 查询报告的顶部包含了一些元数据，包括查询执行的频率、平均并发度，以及该查询性能最差的一次执行在日志文件中的字节偏移值，接下来还有一个表格格式的元数据，包括诸如标准差一类的统计信息[注9]。

接下来的部分是响应时间的直方图。有趣的是，可以看到上面这个查询在Query_time distribution部分的直方图上有两个明显的高峰，大部分情况下执行都需要几百毫秒，但在快三个数量级的部分也有一个明显的尖峰，几百微秒就能执行完成。如果这是Percona Server的记录，那么在查询日志中还会有更多丰富的属性，可以对查询进行切片分析到底发生了什么。比如可能是因为查询条件传递了不同的值，而这些值的分布很不均衡，导致服务器选择了不同的索引；或者是由于查询缓存命中等。在实际系统中，这种有两个尖峰的直方图的情况很少见，尤其是对于简单的查询，查询越简单执行计划也越稳定。

在细节报告的最后部分是方便复制、粘贴到终端去检查表的模式和状态的语句，以及完

注9： 这里已经是尽可能地简化描述了，实际上Percona Server的查询日志报告会包含更多细节信息，可以帮助理解为什么某条查询花费了144ms去获取一行数据，这个时间实在是太长了。

整的可用于 EXPLAIN 分析执行计划的语句。EXPLAIN 分析的语句要求所有的条件是文本值而不是“指纹”替代符，所以是真正可直接执行的语句。在本例中是执行时间最长的一条实际的查询。

确定需要优化的查询后，可以利用这个报告迅速地检查查询的执行情况。这个工具我们经常使用，并且会根据使用的情况不断进行修正以帮助提升工具的可用性和效率，强烈建议大家都能熟练使用它。MySQL 本身在未来或许也会有更多复杂的测量点和剖析工具，但在本书写作时，通过慢查询日志记录查询或者使用 *pt-query-digest* 分析 *tcpdump* 的结果，是可以找到的最好的两种方式。

3.3.2 剖析单条查询

在定位到需要优化的单条查询后，可以针对此查询“钻取”更多的信息，确认为什么会花费这么长的时间执行，以及需要如何去优化。关于如何优化查询的技术将在本书后续的一些章节讨论，在此之前还需要介绍一些相关的背景知识。本章的主要目的是介绍如何方便地测量查询执行的各部分花费了多少时间，有了这些数据才能决定采用何种优化技术。

不幸的是，MySQL 目前大多数的测量点对于剖析查询都没有什么帮助。当然这种状况正在改善，但在本书写作之际，大多数生产环境的服务器还没有使用包含最新剖析特性的版本。所以在实际应用中，除了 SHOW STATUS、SHOW PROFILE、检查慢查询日志的条 85
目（这还要求必须是 Percona Server，官方 MySQL 版本的慢查询日志缺失了很多附加信息）这三种方法外就没有什么更好的办法了。下面将逐一演示如何使用这三种方法来剖析单条查询，看看每一种方法是如何显示查询的执行情况的。

使用 SHOW PROFILE

SHOW PROFILE 命令是在 MySQL 5.1 以后的版本中引入的，来源于开源社区中的 Jeremy Cole 的贡献。这是在本书写作之际唯一一个在 GA 版本中包含的真正的查询剖析工具。默认是禁用的，但可以通过服务器变量在会话（连接）级别动态地修改。

```
mysql> SET profiling = 1;
```

然后，在服务器上执行的所有语句，都会测量其耗费的时间和其他一些查询执行状态变更相关的数据。这个功能有一定的作用，而且最初的设计功能更强大，但未来版本中可能会被 Performance Schema 所取代。尽管如此，这个工具最有用的作用还是在语句执行期间剖析服务器的具体工作。

当一条查询提交给服务器时，此工具会记录剖析信息到一张临时表，并且给查询赋予一

个从 1 开始的整数标识符。下面是对 Sakila 样本数据库的一个视图的剖析结果[注10]：

```
mysql> SELECT * FROM sakila.nicer_but_slower_film_list;
[query results omitted]
997 rows in set (0.17 sec)
```

该查询返回了 997 行记录，花费了大概 1/6 秒。下面看一下 SHOW PROFILES 有什么结果：

```
mysql> SHOW PROFILES;
+----------+------------+-----------------------------------------------+
| Query_ID | Duration   | Query                                         |
+----------+------------+-----------------------------------------------+
|        1 | 0.16767900 | SELECT * FROM sakila.nicer_but_slower_film_list |
+----------+------------+-----------------------------------------------+
```

首先可以看到的是以很高的精度显示了查询的响应时间，这很好。MySQL 客户端显示的时间只有两位小数，对于一些执行得很快的查询这样的精度是不够的。下面继续看接下来的输出：

86

```
mysql> SHOW PROFILE FOR QUERY 1;
+----------------------+----------+
| Status               | Duration |
+----------------------+----------+
| starting             | 0.000082 |
| Opening tables       | 0.000459 |
| System lock          | 0.000010 |
| Table lock           | 0.000020 |
| checking permissions | 0.000005 |
| checking permissions | 0.000004 |
| checking permissions | 0.000003 |
| checking permissions | 0.000004 |
| checking permissions | 0.000560 |
| optimizing           | 0.000054 |
| statistics           | 0.000174 |
| preparing            | 0.000059 |
| Creating tmp table   | 0.000463 |
| executing            | 0.000006 |
| Copying to tmp table | 0.090623 |
| Sorting result       | 0.011555 |
| Sending data         | 0.045931 |
| removing tmp table   | 0.004782 |
| Sending data         | 0.000011 |
| init                 | 0.000022 |
| optimizing           | 0.000005 |
| statistics           | 0.000013 |
| preparing            | 0.000008 |
| executing            | 0.000004 |
| Sending data         | 0.010832 |
| end                  | 0.000008 |
| query end            | 0.000003 |
| freeing items        | 0.000017 |
| removing tmp table   | 0.000010 |
```

注 10： 整个视图太长，无法在书中全部打印出来，但 Sakila 数据库可以从 MySQL 网站上下载到。

```
| freeing items        | 0.000042 |
| removing tmp table   | 0.001098 |
| closing tables       | 0.000013 |
| logging slow query   | 0.000003 |
| logging slow query   | 0.000789 |
| cleaning up          | 0.000007 |
+----------------------+----------+
```

剖析报告给出了查询执行的每个步骤及其花费的时间，看结果很难快速地确定哪个步骤花费的时间最多。因为输出是按照执行顺序排序，而不是按花费的时间排序的——而实际上我们更关心的是花费了多少时间，这样才能知道哪些开销比较大。但不幸的是无法通过诸如 ORDER BY 之类的命令重新排序。假如不使用 SHOW PROFILE 命令而是直接查询 INFORMATION_SCHEMA 中对应的表，则可以按照需要格式化输出：

87

```
mysql> SET @query_id = 1;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT STATE, SUM(DURATION) AS Total_R,
    ->    ROUND(
    ->       100 * SUM(DURATION) /
    ->          (SELECT SUM(DURATION)
    ->           FROM INFORMATION_SCHEMA.PROFILING
    ->           WHERE QUERY_ID = @query_id
    ->       ), 2) AS Pct_R,
    ->    COUNT(*) AS Calls,
    ->    SUM(DURATION) / COUNT(*) AS "R/Call"
    -> FROM INFORMATION_SCHEMA.PROFILING
    -> WHERE QUERY_ID = @query_id
    -> GROUP BY STATE
    -> ORDER BY Total_R DESC;
+----------------------+----------+-------+-------+--------------+
| STATE                | Total_R  | Pct_R | Calls | R/Call       |
+----------------------+----------+-------+-------+--------------+
| Copying to tmp table | 0.090623 | 54.05 |     1 | 0.0906230000 |
| Sending data         | 0.056774 | 33.86 |     3 | 0.0189246667 |
| Sorting result       | 0.011555 |  6.89 |     1 | 0.0115550000 |
| removing tmp table   | 0.005890 |  3.51 |     3 | 0.0019633333 |
| logging slow query   | 0.000792 |  0.47 |     2 | 0.0003960000 |
| checking permissions | 0.000576 |  0.34 |     5 | 0.0001152000 |
| Creating tmp table   | 0.000463 |  0.28 |     1 | 0.0004630000 |
| Opening tables       | 0.000459 |  0.27 |     1 | 0.0004590000 |
| statistics           | 0.000187 |  0.11 |     2 | 0.0000935000 |
| starting             | 0.000082 |  0.05 |     1 | 0.0000820000 |
| preparing            | 0.000067 |  0.04 |     2 | 0.0000335000 |
| freeing items        | 0.000059 |  0.04 |     2 | 0.0000295000 |
| optimizing           | 0.000059 |  0.04 |     2 | 0.0000295000 |
| init                 | 0.000022 |  0.01 |     1 | 0.0000220000 |
| Table lock           | 0.000020 |  0.01 |     1 | 0.0000200000 |
| closing tables       | 0.000013 |  0.01 |     1 | 0.0000130000 |
| System lock          | 0.000010 |  0.01 |     1 | 0.0000100000 |
| executing            | 0.000010 |  0.01 |     2 | 0.0000050000 |
| end                  | 0.000008 |  0.00 |     1 | 0.0000080000 |
| cleaning up          | 0.000007 |  0.00 |     1 | 0.0000070000 |
| query end            | 0.000003 |  0.00 |     1 | 0.0000030000 |
+----------------------+----------+-------+-------+--------------+
```

效果好多了！通过这个结果可以很容易看到查询时间太长主要是因为花了一大半的时间在将数据复制到临时表这一步。那么优化就要考虑如何改写查询以避免使用临时表，或者提升临时表的使用效率。第二个消耗时间最多的是“发送数据（Sending data)”，这个状态代表的原因非常多，可能是各种不同的服务器活动，包括在关联时搜索匹配的行记录等，这部分很难说能优化节省多少消耗的时间。另外也要注意到“结果排序（Sorting result)”花费的时间占比非常低，所以这部分是不值得去优化的。这是一个比较典型的问题，所以一般我们都不建议用户在“优化排序缓冲区（tuning sort buffer)”或者类似的活动上花时间。

尽管剖析报告能帮助我们定位到哪些活动花费了最多的时间，但并不会告诉我们为什么会这样。要弄清楚为什么复制数据到临时表要花费这么多时间，就需要深入下去，继续剖析这一步的子任务。

88 使用 SHOW STATUS

MySQL 的 SHOW STATUS 命令返回了一些计数器。既有服务器级别的全局计数器，也有基于某个连接的会话级别的计数器。例如其中的 Queries[注11] 在会话开始时为 0，每提交一条查询增加 1。如果执行 SHOW GLOBAL STATUS（注意到新加的 GLOBAL 关键字)，则可以查看服务器级别的从服务器启动时开始计算的查询次数统计。不同计数器的可见范围不一样，不过全局的计数器也会出现在 SHOW STATUS 的结果中，容易被误认为是会话级别的，千万不要搞迷糊了。在使用这个命令的时候要注意几点，就像前面所讨论的，收集合适级别的测量值是很关键的。如果打算优化从某些特定连接观察到的东西，测量的却是全局级别的数据，就会导致混乱。MySQL 官方手册中对所有的变量是会话级还是全局级做了详细的说明。

SHOW STATUS 是一个有用的工具，但并不是一款剖析工具[注12]。SHOW STATUS 的大部分结果都只是一个计数器，可以显示某些活动如读索引的频繁程度，但无法给出消耗了多少时间。SHOW STATUS 的结果中只有一条指的是操作的时间（Innodb_row_lock_time)，而且只能是全局级的，所以还是无法测量会话级别的工作。

尽管 SHOW STATUS 无法提供基于时间的统计，但对于在执行完查询后观察某些计数器的值还是有帮助的。有时候可以猜测哪些操作代价较高或者消耗的时间较多。最有用的计数器包括句柄计数器（handler counter)、临时文件和表计数器等。在附录 B 中会对此做更详细的解释。下面的例子演示了如何将会话级别的计数器重置为 0，然后查询前面（“使用 SHOW PROFILE”一节）提到的视图，再检查计数器的结果：

注 11： 原文用的 Queries，实际上这里有点问题，虽然文档上也说这个参数是会话级的，但在 MySQL 5.1/5.5 多个版本中实际查询时发现其是全局级别的。——译者注

注 12： 如果你有本书的第二版，可能会注意到我们正在彻底改变这一点。

```
mysql> FLUSH STATUS;
mysql> SELECT * FROM sakila.nicer_but_slower_film_list;
[query results omitted]
mysql> SHOW STATUS WHERE Variable_name LIKE 'Handler%'
        OR Variable_name LIKE 'Created%';
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| Created_tmp_disk_tables    | 2     |
| Created_tmp_files          | 0     |
| Created_tmp_tables         | 3     |
| Handler_commit             | 1     |
| Handler_delete             | 0     |
| Handler_discover           | 0     |
| Handler_prepare            | 0     |
| Handler_read_first         | 1     |
| Handler_read_key           | 7483  |
| Handler_read_next          | 6462  |
| Handler_read_prev          | 0     |
| Handler_read_rnd           | 5462  |
| Handler_read_rnd_next      | 6478  |
| Handler_rollback           | 0     |
| Handler_savepoint          | 0     |
| Handler_savepoint_rollback | 0     |
| Handler_update             | 0     |
| Handler_write              | 6459  |
+----------------------------+-------+
```

89

从结果可以看到该查询使用了三个临时表，其中两个是磁盘临时表，并且有很多的没有用到索引的读操作（Handler_read_rnd_next）。假设我们不知道这个视图的具体定义，仅从结果来推测，这个查询有可能是做了多表关联（join）查询，并且没有合适的索引，可能是其中一个子查询创建了临时表，然后和其他表做联合查询。而用于保存子查询结果的临时表没有索引，如此大致可以解释这样的结果。

使用这个技术的时候，要注意 SHOW STATUS 本身也会创建一个临时表，而且也会通过句柄操作访问此临时表，这会影响到 SHOW STATUS 结果中对应的数字，而且不同的版本可能行为也不尽相同。比较前面通过 SHOW PROFILES 获得的查询的执行计划的结果来看，至少临时表的计数器多加了 2。

你可能会注意到通过 EXPLAIN 查看查询的执行计划也可以获得大部分相同的信息，但 EXPLAIN 是通过估计得到的结果，而通过计数器则是实际的测量结果。例如，EXPLAIN 无法告诉你临时表是否是磁盘表，这和内存临时表的性能差别是很大的。附录 D 包含更多关于 EXPLAIN 的内容。

使用慢查询日志

那么针对上面这样的查询语句，Percona Server 对慢查询日志做了哪些改进？下面是“使用 SHOW PROFILE”一节演示过的相同的查询执行后抓取到的结果：

```
# Time: 110905 17:03:18
# User@Host: root[root] @ localhost [127.0.0.1]
# Thread_id: 7  Schema: sakila  Last_errno: 0  Killed: 0
# Query_time: 0.166872  Lock_time: 0.000552  Rows_sent: 997  Rows_examined: 24861
  Rows_affected: 0  Rows_read: 997
# Bytes_sent: 216528  Tmp_tables: 3  Tmp_disk_tables: 2  Tmp_table_sizes: 11627188
# InnoDB_trx_id: 191E
# QC_Hit: No  Full_scan: Yes  Full_join: No  Tmp_table: Yes  Tmp_table_on_disk: Yes
# Filesort: Yes  Filesort_on_disk: No  Merge_passes: 0
#   InnoDB_IO_r_ops: 0  InnoDB_IO_r_bytes: 0  InnoDB_IO_r_wait: 0.000000
#   InnoDB_rec_lock_wait: 0.000000  InnoDB_queue_wait: 0.000000
#   InnoDB_pages_distinct: 20
# PROFILE_VALUES ... Copying to tmp table: 0.090623... [omitted]
SET timestamp=1315256598;
SELECT * FROM sakila.nicer_but_slower_film_list;
```

90 从这里看到查询确实一共创建了三个临时表，其中两个是磁盘临时表。而 SHOW PROFILE 看起来则隐藏了信息（可能是由于服务器执行查询的方式有不一样的地方造成的）。这里为了方便阅读，对结果做了简化。但最后对该查询执行 SHOW PROFILE 的数据也会写入到日志中，所以在 Percona Server 中甚至可以记录 SHOW PROFILE 的细节信息。

另外也可以看到，慢查询日志中详细记录的条目包含了 SHOW PROFILE 和 SHOW STATUS 所有的输出，并且还有更多的信息。所以通过 *pt-query-digest* 发现“坏”查询后，在慢查询日志中可以获得足够有用的信息。查看 *pt-query-digest* 的报告时，其标题部分一般会有如下输出：

```
# Query 1: 0 QPS, 0x concurrency, ID 0xEE758C5E0D7EADEE at byte 3214 _____
```

可以通过这里的字节偏移值（3214）直接跳转到日志的对应部分，例如用下面这样的命令即可：

```
tail -c +3214 /path/to/query.log | head -n100
```

这样就可以直接跳转到细节部分了。另外，*pt-query-digest* 能够处理 Percona Server 在慢查询日志中增加的所有键值对，并且会自动在报告中打印更多的细节信息。

使用 Performance Schema

在本书写作之际，在 MySQL 5.5 中新增的 Performance Schema 表还不支持查询级别的剖析信息。Performance Schema 还是非常新的特性，并且还在快速开发中，未来的版本中将会包含更多的功能。尽管如此，MySQL 5.5 的初始版本已经包含了很多有趣的信息。例如，下面的查询显示了系统中等待的主要原因：

```
mysql> SELECT event_name, count_star, sum_timer_wait
    -> FROM events_waits_summary_global_by_event_name
    -> ORDER BY sum_timer_wait DESC LIMIT 5;
+----------------------------------------+------------+------------------+
| event_name                             | count_star | sum_timer_wait   |
+----------------------------------------+------------+------------------+
| innodb_log_file                        | 205438     | 2552133070220355 |
| Query_cache::COND_cache_status_changed | 8405302    | 2259497326493034 |
| Query_cache::structure_guard_mutex     | 55769435   |  361568224932147 |
| innodb_data_file                       | 62423      |  347302500600411 |
| dict_table_stats                       | 15330162   |   53005067680923 |
+----------------------------------------+------------+------------------+
```

目前还有一些限制，使得 Performance Schema 还无法被当作一个通用的剖析工具。首先，它还无法提供查询执行阶段的细节信息和计时信息，而前面提供的很多现有的工具都已经能做到这些了。其次，还没有经过长时间、大规模使用的验证，并且自身的开销也还比较大，多数比较保守的用户还对此持有疑问（不过有理由相信这些问题很快都会被修复的）。

最后，对大多数用户来说，直接通过 Performance Schema 的裸数据获得有用的结果相对来说过于复杂和底层。到目前为止实现的这个特性，主要是为了测量当为提升服务器性能而修改 MySQL 源代码时使用，包括等待和互斥锁。MySQL 5.5 中的特性对于高级用户也很有价值，而不仅仅为开发者使用，但还是需要开发一些前端工具以方便用户使用和分析结果。目前就只能通过写一些复杂的语句去查询大量的元数据表的各种列。这在使用过程中需要花很多时间去熟悉和理解。 91

在 MySQL 5.6 或者以后的版本中，Performance Schema 将会包含更多的功能，再加上一些方便使用的工具，这样就更"爽"了。而且 Oracle 将其实现成表的形式，可以通过 SQL 访问，这样用户可以方便地访问有用的数据。但其目前还无法立即取代慢查询日志等其他工具用于服务器和查询的性能优化。

3.3.3 使用性能剖析

当获得服务器或者查询的剖析报告后，怎么使用？好的剖析报告能够将潜在的问题显示出来，但最终的解决方案还需要用户来决定（尽管报告可能会给出建议）。优化查询时，用户需要对服务器如何执行查询有较深的了解。剖析报告能够尽可能多地收集需要的信息、给出诊断问题的正确方向，以及为其他诸如 EXPLAIN 等工具提供基础信息。这里只是先引出话题，后续章节将继续讨论。

尽管一个拥有完整测量信息的剖析报告可以让事情变得简单，但现有系统通常都没有完美的测量支持。从前面的例子来说，我们虽然推断出是临时表和没有索引的读导致查询

的响应时间过长，但却没有明确的证据。因为无法测量所有需要的信息，或者测量的范围不正确，有些问题就很难解决。例如，可能没有集中在需要优化的地方测量，而是测量了服务器层面的活动；或者测量的是查询开始之前的计数器，而不是查询开始后的数据。

也有其他的可能性。设想一下正在分析慢查询日志，发现了一个很简单的查询正常情况下都非常快，却有几次非常不合理地执行了很长时间。手工重新执行一遍，发现也非常快，然后使用 EXPLAIN 查询其执行计划，也正确地使用了索引。然后尝试修改 WHERE 条件中使用不同的值，以排除缓存命中的可能，也没有发现有什么问题，这可能是什么原因呢？

92 如果使用官方版本的 MySQL，慢查询日志中没有执行计划或者详细的时间信息，对于偶尔记录到的这几次查询异常慢的问题，很难知道其原因在哪里，因为信息有限。可能是系统中有其他东西消耗了资源，比如正在备份，也可能是某种类型的锁或者争用阻塞了查询的进度。这种间歇性的问题将在下一节详细讨论。

3.4 诊断间歇性问题

间歇性的问题比如系统偶尔停顿或者慢查询，很难诊断。有些幻影问题只在没有注意到的时候才发生，而且无法确认如何重现，诊断这样的问题往往要花费很多时间，有时候甚至需要好几个月。在这个过程中，有些人会尝试以不断试错的方式来诊断，有时候甚至会想要通过随机地改变一些服务器的设置来侥幸地找到问题。

尽量不要使用试错的方式来解决问题。这种方式有很大的风险，因为结果可能变得更坏。这也是一种令人沮丧且低效的方式。如果一时无法定位问题，可能是测量的方式不正确，或者测量的点选择有误，或者使用的工具不合适（也可能是缺少现成的工具，我们已经开发过工具来解决各个系统不透明导致的问题，包括从操作系统到 MySQL 都有）。

为了演示为什么要尽量避免试错的诊断方式，下面列举了我们认为已经解决的一些间歇性数据库性能问题的实际案例：

- 应用通过 *curl* 从一个运行得很慢的外部服务来获取汇率报价的数据。
- *memcached* 缓存中的一些重要条目过期，导致大量请求落到 MySQL 以重新生成缓存条目。
- DNS 查询偶尔会有超时现象。
- 可能是由于互斥锁争用，或者内部删除查询缓存的算法效率太低的缘故，MySQL 的查询缓存有时候会导致服务有短暂的停顿。

- 当并发度超过某个阈值时，InnoDB 的扩展性限制导致查询计划的优化需要很长的时间。

从上面可以看到，有些问题确实是数据库的原因，也有些不是。只有在问题发生的地方通过观察资源的使用情况，并尽可能地测量出数据，才能避免在没有问题的地方耗费精力。

下面不再多费口舌说明试错的问题，而是给出我们解决间歇性问题的方法和工具，这才是“王道”。 93

3.4.1 单条查询问题还是服务器问题

发现问题的蛛丝马迹了吗？如果有，则首先要确认这是单条查询的问题，还是服务器的问题。这将为解决问题指出正确的方向。如果服务器上所有的程序都突然变慢，又突然都变好，每一条查询也都变慢了，那么慢查询可能就不一定是原因，而是由于其他问题导致的结果。反过来说，如果服务器整体运行没有问题，只有某条查询偶尔变慢，就需要将注意力放到这条特定的查询上面。

服务器的问题非常常见。在过去几年，硬件的能力越来越强，配置 16 核或者更多 CPU 的服务器成了标配，MySQL 在 SMP 架构的机器上的可扩展性限制也就越来越显露出来。尤其是较老的版本，其问题更加严重，而目前生产环境中的老版本还非常多。新版本 MySQL 依然也还有一些扩展性限制，但相比老版本已经没有那么严重，而且出现的频率相对小很多，只是偶尔能碰到。这是好消息，也是坏消息：好消息是很少会碰到这个问题；坏消息则是一旦碰到，则需要对 MySQL 内部机制更加了解才能诊断出来。当然，这也意味着很多问题可以通过升级到 MySQL 新版本来解决[注13]。

那么如何判断是单条查询问题还是服务器问题呢？如果问题不停地周期性出现，那么可以在某次活动中观察到；或者整夜运行脚本收集数据，第二天来分析结果。大多数情况下都可以通过三种技术来解决，下面将一一道来。

使用 SHOW GLOBAL STATUS

这个方法实际上就是以较高的频率比如一秒执行一次 `SHOW GLOBAL STATUS` 命令捕获数据，问题出现时，则可以通过某些计数器（比如 `Threads_running`、`Threads_connected`、`Questions` 和 `Queries`）的“尖刺”或者“凹陷”来发现。这个方法比较简单，所有人都可以使用（不需要特殊的权限），对服务器的影响也很小，所以是一个花费时间不多却能很好地了解问题的好方法。下面是示例命令及其输出：

注 13：再次强调，在没有足够的理由确信这是解决办法之前，不要随便去做升级操作。

```
$ mysqladmin ext -i1 | awk '
    /Queries/{q=$4-qp;qp=$4}
    /Threads_connected/{tc=$4}
    /Threads_running/{printf "%5d %5d %5d\n", q, tc, $4}'
2147483647   136     7
  798   136     7
  767   134     9
  828   134     7
94  683   134     7
  784   135     7
  614   134     7
  108   134    24
  187   134    31
  179   134    28
 1179   134     7
 1151   134     7
 1240   135     7
 1000   135     7
```

这个命令每秒捕获一次 SHOW GLOBAL STATUS 的数据，输出给 *awk* 计算并输出每秒的查询数、Threads_connected 和 Threads_running（表示当前正在执行查询的线程数）。这三个数据的趋势对于服务器级别偶尔停顿的敏感性很高。一般发生此类问题时，根据原因的不同和应用连接数据库方式的不同，每秒的查询数一般会下跌，而其他两个则至少有一个会出现尖刺。在这个例子中，应用使用了连接池，所以 Threads_connected 没有变化。但正在执行查询的线程数明显上升，同时每秒的查询数相比正常数据有严重的下跌。

如何解析这个现象呢？凭猜测有一定的风险。但在实践中有两个原因的可能性比较大。其中之一是服务器内部碰到了某种瓶颈，导致新查询在开始执行前因为需要获取老查询正在等待的锁而造成堆积。这一类的锁一般也会对应用服务器造成后端压力，使得应用服务器也出现排队问题。另外一个常见的原因是服务区突然遇到了大量查询请求的冲击，比如前端的 *memcached* 突然失效导致的查询风暴。

这个命令每秒输出一行数据，可以运行几个小时或者几天，然后将结果绘制成图形，这样就可以方便地发现是否有趋势的突变。如果问题确实是间歇性的，发生的频率又较低，也可以根据需要尽可能长时间地运行此命令，直到发现问题再回头来看输出结果。大多数情况下，通过输出结果都可以更明确地定位问题。

使用 SHOW PROCESSLIST

这个方法是通过不停地捕获 SHOW PROCESSLIST 的输出，来观察是否有大量线程处于不正常的状态或者有其他不正常的特征。例如查询很少会长时间处于“statistics”状态，这个状态一般是指服务器在查询优化阶段如何确定表关联的顺序——通常都是非常快的。另外，也很少会见到大量线程报告当前连接用户是“未经验证的用户（Unauthenticated

user)”，这只是在连接握手的中间过程中的状态，当客户端等待输入用于登录的用户信息的时候才会出现。

使用 SHOW PROCESSLIST 命令时，在尾部加上 \G 可以垂直的方式输出结果，这很有用，因为这样会将每一行记录的每一列都单独输出为一行，这样可以方便地使用 *sort|uniq|sort* 一类的命令来计算某个列值出现的次数：

```
$ mysql -e 'SHOW PROCESSLIST\G' | grep State: | sort | uniq -c | sort -rn    95
    744   State:
     67   State: Sending data
     36   State: freeing items
      8   State: NULL
      6   State: end
      4   State: Updating
      4   State: cleaning up
      2   State: update
      1   State: Sorting result
      1   State: logging slow query
```

如果要查看不同的列，只需要修改 *grep* 的模式即可。在大多数案例中，State 列都非常有用。从这个例子的输出中可以看到，有很多线程处于查询执行的结束部分的状态，包括“freeing items”、“end”、“cleaning up”和“logging slow query”。事实上，在案例中的这台服务器上，同样模式或类似的输出采样出现了很多次。大量的线程处于“freeing items”状态是出现了大量有问题查询的很明显的特征和指示。

用这种技术查找问题，上面的命令行不是唯一的方法。如果 MySQL 服务器的版本较新，也可以直接查询 INFORMATION_SCHEMA 中的 PROCESSLIST 表；或者使用 *innotop* 工具以较高的频率刷新，以观察屏幕上出现的不正常查询堆积。上面演示的这个例子是由于 InnoDB 内部的争用和脏块刷新所导致，但有时候原因可能比这个要简单得多。一个经典的例子是很多查询处于“Locked”状态，这是 MyISAM 的一个典型问题，它的表级别锁定，在写请求较多时，可能迅速导致服务器级别的线程堆积。

使用查询日志

如果要通过查询日志发现问题，需要开启慢查询日志并在全局级别设置 long_query_time 为 0，并且要确认所有的连接都采用了新的设置。这可能需要重置所有连接以使新的全局设置生效；或者使用 Percona Server 的一个特性，可以在不断开现有连接的情况下动态地使设置强制生效。

如果因为某些原因，不能设置慢查询日志记录所有的查询，也可以通过 *tcpdump* 和 *pt-query-digest* 工具来模拟替代。要注意找到吞吐量突然下降时间段的日志。查询是在完成阶段才写入到慢查询日志的，所以堆积会造成大量查询处于完成阶段，直到阻塞其他查

询的资源占用者释放资源后，其他的查询才能执行完成。这种行为特征的一个好处是，当遇到吞吐量突然下降时，可以归咎于吞吐量下降后完成的第一个查询（有时候也不一定是第一个查询。当某些查询被阻塞时，其他查询可以不受影响继续运行，所以不能完全依赖这个经验）。

再重申一次，好的工具可以帮助诊断这类问题，否则要人工去几百 GB 的查询日志中找原因。下面的例子只有一行代码，却可以根据 MySQL 每秒将当前时间写入日志中的模式统计每秒的查询数量：

96

```
$ awk '/^# Time:/{print $3, $4, c;c=0}/^# User/{c++}' slow-query.log
080913 21:52:17 51
080913 21:52:18 29
080913 21:52:19 34
080913 21:52:20 33
080913 21:52:21 38
080913 21:52:22 15
080913 21:52:23 47
080913 21:52:24 96
080913 21:52:25 6
080913 21:52:26 66
080913 21:52:27 37
080913 21:52:28 59
```

从上面的输出可以看到有吞吐量突然下降的情况发生，而且在下降之前还有一个突然的高峰，仅从这个输出而不去查询当时的详细信息很难确定发生了什么，但应该可以说这个突然的高峰和随后的下降一定有关联。不管怎么说，这种现象都很奇怪，值得去日志中挖掘该时间段的详细信息（实际上通过日志的详细信息，可以发现突然的高峰时段有很多连接被断开的现象，可能是有一台应用服务器重启导致的。所以不是所有的问题都是 MySQL 的问题）。

理解发现的问题（Making sense of the findings）

可视化的数据最具有说服力。上面只演示了很少的几个例子，但在实际情况中，利用上面的工具诊断时可能产生大量的输出结果。可以选择用 *gnuplot* 或 *R*，或者其他绘图工具将结果绘制成图形。这些绘图工具速度很快，比电子表格要快得多，而且可以对图上的一些异常的地方进行缩放，这比在终端中通过滚动条翻看文字要好用得多，除非你是“黑客帝国”中的矩阵观察者[注14]。

我们建议诊断问题时先使用前两种方法：SHOW STATUS 和 SHOW PROCESSLIST。这两种方法的开销很低，而且可以通过简单的 shell 脚本或者反复执行的查询来交互式地收集数据。分析慢查询日志则相对要困难一些，经常会发现一些蛛丝马迹，但仔细去研究时可

注 14：到目前为止我们还没发现红衣女，如果发现了，一定会让你知道的。

能又消失了。这样我们很容易会认为其实没有问题。

发现输出的图形异常意味着什么？通常来说可能是查询在某个地方排队了，或者某种查询的量突然飙升了。接下来的任务就是找出这些原因。

3.4.2 捕获诊断数据

97

当出现间歇性问题时，需要尽可能多地收集所有数据，而不只是问题出现时的数据。虽然这样会收集大量的诊断数据，但总比真正能够诊断问题的数据没有被收集到的情况要好。

在开始之前，需要搞清楚两件事：

1. 一个可靠且实时的“触发器”，也就是能区分什么时候问题出现的方法。
2. 一个收集诊断数据的工具。

诊断触发器

触发器非常重要。这是在问题出现时能够捕获数据的基础。有两个常见的问题可能导致无法达到预期的结果：误报（false positive）或者漏检（false negative）。误报是指收集了很多诊断数据，但期间其实没有发生问题，这可能浪费时间，而且令人沮丧。而漏检则指在问题出现时没有捕获到数据，错失了机会，一样地浪费时间。所以在开始收集数据前多花一点时间来确认触发器能够真正地识别问题是划算的。

那么好的触发器的标准是什么呢？像前面的例子展示的，`Threads_running` 的趋势在出现问题时会比较敏感，而没有问题时则比较平稳。另外 `SHOW PROCESSLIST` 中线程的异常状态尖峰也是个不错的指标。当然除此之外还有很多的方法，包括 `SHOW INNODB STATUS` 的特定输出、服务器的平均负载尖峰等。关键是找到一些能和正常时的阈值进行比较的指标。通常情况下这是一个计数，比如正在运行的线程的数量、处于“freeing items”状态的线程的数量等。当要计算线程某个状态的数量时，*grep* 的 *-c* 选项非常有用：

```
$ mysql -e 'SHOW PROCESSLIST\G' | grep -c "State: freeing items"
36
```

选择一个合适的阈值很重要，既要足够高，以确保在正常时不会被触发；又不能太高，要确保问题发生时不会错过。另外要注意，要在问题开始时就捕获数据，就更不能将阈值设置得太高。问题持续上升的趋势一般会导致更多的问题发生，如果在问题导致系统快要崩溃时才开始捕获数据，就很难诊断到最初的根本原因。如果可能，在问题还是涓涓细流的时候就要开始收集数据，而不要等到波涛汹涌才开始。举个例子，`Threads_connected` 偶尔出现非常高的尖峰值，在几分钟时间内会从 100 冲到 5 000 或者更高，

所以设置阈值为 4 999 也可以捕获到问题，但为什么非要等到这么高的时候才收集数据呢？如果在正常时该值一般不超过 150，将阈值设置为 200 或者 300 会更好。

98 回到前面关于 Threads_running 的例子，正常情况下的并发度不超过 10。但是阈值设置为 10 并不是一个好注意，很可能会导致很多误报。即使设置为 15 也不够，可能还是会有很多正常的波动会到这个范围。当并发运行线程到 15 的时候可能也会有少量堆积的情况，但可能还没到问题的引爆点。但也应该在糟糕到一眼就能看出问题前就清晰地识别出来，对于这个例子，我们建议阀值可以设置为 20。

我们当然希望在问题确实发生时能捕获到数据，但有时候也需要稍微等待一下以确保不是误报或者短暂的尖峰。所以，最后的触发条件可以这样设置：每秒监控状态值，如果 Threads_running 连续 5 秒超过 20，就开始收集诊断数据（顺便说一句，我们的例子中问题只持续了 3 秒就消失了，这是为了使例子简单而设置的。3 秒的故障不容易诊断，而我们碰到过的大部分问题持续时间都会更长一些）。

所以我们需要利用一种工具来监控服务器，当达到触发条件时能收集数据。当然可以自己编写脚本来实现，不过不用那么麻烦，Percona Toolkit 中的 *pt-stalk* 就是为这种情况设计的。这个工具有很多有用的特性，只要碰到过类似问题就会明白这些特性的必要性。例如，它会监控磁盘的可用空间，所以不会因为收集太多的数据将空间耗尽而导致服务器崩溃。如果之前碰到过这样的情况，你就会理解这一点了。

pt-stalk 的用法很简单。可以配置需要监控的变量、阈值、检查的频率等。还支持一些比实际需要更多的花哨特性，但在这个例子中有这些已经足够了。在使用之前建议先阅读附带的文档。*pt-stalk* 还依赖于另外一个工具执行真正的收集工作，接下来会讨论。

需要收集什么样的数据

现在已经确定了诊断触发器，可以开始启动一些进程来收集数据了。但需要收集什么样的数据呢？就像前面说的，答案是尽可能收集所有能收集的数据，但只在需要的时间段内收集。包括系统的状态、CPU 利用率、磁盘使用率和可用空间、ps 的输出采样、内存利用率，以及可以从 MySQL 获得的信息，如 SHOW STATUS、SHOW PROCESSLIST 和 SHOW INNODB STATUS。这些在诊断问题时都需要用到（可能还会有更多）。

执行时间包括用于工作的时间和等待的时间。当一个未知问题发生时，一般来说有两种可能：服务器需要做大量的工作，从而导致大量消耗 CPU；或者在等待某些资源被释放。所以需要用不同的方法收集诊断数据，来确认是何种原因：剖析报告用于确认是否有太多工作，而等待分析则用于确认是否存在大量等待。如果是未知的问题，怎么知道将精力集中在哪个方面呢？没有更好的办法，所以只能两种数据都尽量收集。

在 GNU/Linux 平台，可用于服务器内部诊断的一个重要工具是 *oprofile*。后面会展示一些例子。也可以使用 *strace* 剖析服务器的系统调用，但在生产环境中使用它有一定的风险。后面还会继续讨论它。如果要剖析查询，可以使用 *tcpdump*。大多数 MySQL 版本无法方便地打开和关闭慢查询日志，此时可以通过监听 TCP 流量来模拟。另外，网络流量在其他一些分析中也非常有用。 99

对于等待分析，常用的方法是 GDB 的堆栈跟踪[注15]。MySQL 内的线程如果卡在一个特定的地方很长时间，往往都有相同的堆栈跟踪信息。跟踪的过程是先启动 *gdb*，然后附加（attach）到 *mysqld* 进程，将所有线程的堆栈都转储出来。然后可以利用一些简短的脚本将类似的堆栈跟踪信息做汇总，再利用 *sort|uniq|sort* 的"魔法"排序出总计最多的堆栈信息。稍后将演示如何用 *pt-pmp* 工具来完成这个工作。

也可以使用 `SHOW PROCESSLIST` 和 `SHOW INNODB STATUS` 的快照信息观察线程和事务的状态来进行等待分析。这些方法都不完美，但实践证明还是非常有帮助的。

收集所有的数据听起来工作量很大。或许读者之前已经做过类似的事情，但我们提供的工具可以提供一些帮助。这个工具名为 *pt-collect*，也是 Percona Toolkit 中的一员。*pt-collect* 一般通过 *pt-stalk* 来调用。因为涉及很多重要数据的收集，所以需要用 *root* 权限来运行。默认情况下，启动后会收集 30 秒的数据，然后退出。对于大多数问题的诊断来说，这已经足够，但如果有误报（false positive）的问题出现，则可能收集的信息就不够。

这个工具很容易下载到，并且不需要任何配置，配置都是通过 *pt-stalk* 进行的。系统中最好安装 *gdb* 和 *oprofile*，然后在 *pt-stalk* 中配置使用。另外 *mysqld* 也需要有调试符号信息[注16]。当触发条件满足时，*pt-collect* 会很好地收集完整的数据。它也会在目录中创建时间戳文件。在本书写作之际，这个工具是基于 GNU/Linux 的，后续会迁移到其他操作系统，这是一个好的开始。

解释结果数据

如果已经正确地设置好触发条件，并且长时间运行 *pt-stalk*，则只需要等待足够长的时间来捕获几次问题，就能够得到大量的数据来进行筛选。从哪里开始最好呢？我们建议先根据两个目的来查看一些东西。第一，检查问题是否真的发生了，因为有很多的样本数 100
据需要检查，如果是误报就会白白浪费大量的时间。第二，是否有非常明显的跳跃性变化。

注 15： 警告：使用 GDB 是有侵入性的。它会暂时造成服务器停顿，尤其是有很多线程的时候，甚至有可能造成崩溃。但有时候收益还是大于风险的。如果服务器本身问题已经严重到无法提供服务了，那么使用 GBD 再造成一些暂停也就无所谓了。

注 16： 有时候为了"优化"而不安装符号信息，实际上这样做不会有多少优化的效果，反而会造成诊断问题更困难。可以使用 *nm* 工具检查是否安装了符号信息，如果没有，则可以通过安装 MySQL 的 *debuginfo* 包来安装。

> 在服务器正常运行时捕获一些样本数据也很重要，而不只是在有问题时捕获数据。这样可以帮助对比确认是否某些样本，或者样本中的某部分数据有异常。例如，在查看进程列表（process list）中查询的状态时，可以回答一些诸如“大量查询处于正在排序结果的状态是不是正常的”的问题。

查看异常的查询或事务的行为，以及异常的服务器内部行为通常都是最有收获的。查询或事务的行为可以显示是否是由于使用服务器的方式导致的问题：性能低下的 SQL 查询、使用不当的索引、设计糟糕的数据库逻辑架构等。通过抓取 TCP 流量或者 SHOW PROCESSLIST 输出，可以获得查询和事务出现的地方，从而知道用户对数据库进行了什么操作。通过服务器的内部行为则可以清楚服务器是否有 bug，或者内部的性能和扩展性是否有问题。这些信息在类似的地方都可以看到，包括在 *oprofile* 或者 *gdb* 的输出中，但要理解则需要更多的经验。

如果遇到无法解释的错误，则最好将收集到的所有数据打包，提交给技术支持人员进行分析。MySQL 的技术支持专家应该能够从数据中分析出原因，详细的数据对于支持人员来说非常重要。另外也可以将 Percona Toolkit 中另外两款工具 *pt-mysql-summary* 和 *pt-summary* 的输出结果打包，这两个工具会输出 MySQL 的状态和配置信息，以及操作系统和硬件的信息。

Percona Toolkit 还提供了一款快速检查收集到的样本数据的工具：*pt-sift*。这个工具会轮流导航到所有的样本数据，得到每个样本的汇总信息。如果需要，也可以钻取到详细信息。使用此工具至少可以少打很多字，少敲很多次键盘。

前面我们演示了状态计数器和线程状态的例子。在本章结束之前，将再给出一些 *oprofile* 和 *gdb* 的输出例子。下面是一个问题服务器上的 *oprofile* 输出，你能找到问题吗？

101

```
samples  %        image name   app name     symbol  name
893793   31.1273  /no-vmlinux  /no-vmlinux  (no symbols)
325733   11.3440  mysqld       mysqld       Query_cache::free_memory_block()
117732    4.1001  libc         libc         (no symbols)
102349    3.5644  mysqld       mysqld       my_hash_sort_bin
 76977    2.6808  mysqld       mysqld       MYSQLparse()
 71599    2.4935  libpthread   libpthread   pthread_mutex_trylock
 52203    1.8180  mysqld       mysqld       read_view_open_now
 46516    1.6200  mysqld       mysqld       Query_cache::invalidate_query_block_list()
 42153    1.4680  mysqld       mysqld       Query_cache::write_result_data()
 37359    1.3011  mysqld       mysqld       MYSQLlex()
 35917    1.2508  libpthread   libpthread   __pthread_mutex_unlock_usercnt
 34248    1.1927  mysqld       mysqld       __intel_new_memcpy
```

如果你的答案是“查询缓存”，那么恭喜你答对了。在这里查询缓存导致了大量的工作，并拖慢了整个服务器。这个问题是一夜之间突然发生的，系统变慢了 50 倍，但这期间

系统没有做过任何其他变更。关闭查询缓存后系统性能恢复了正常。这个例子比较简单地解释了服务器内部行为对性能的影响。

另外一个重要的关于等待分析的性能瓶颈分析工具是 *gdb* 的堆栈跟踪。下面是对一个线程的堆栈跟踪的输出结果，为了便于印刷做了一些格式化：

```
Thread 992 (Thread 0x7f6ee0111910 (LWP 31510)):
#0  0x0000003be560b2f9 in pthread_cond_wait@@GLIBC_2.3.2 () from /libpthread.so.0
#1  0x00007f6ee14f0965 in os_event_wait_low () at os/os0sync.c:396
#2  0x00007f6ee1531507 in srv_conc_enter_innodb () at srv/srv0srv.c:1185
#3  0x00007f6ee14c906a in innodb_srv_conc_enter_innodb () at handler/ha_innodb.cc:609
#4  ha_innodb::index_read () at handler/ha_innodb.cc:5057
#5  0x00000000006538c5 in ?? ()
#6  0x0000000000658029 in sub_select() ()
#7  0x0000000000658e25 in ?? ()
#8  0x00000000006677c0 in JOIN::exec() ()
#9  0x000000000066944a in mysql_select() ()
#10 0x0000000000669ea4 in handle_select() ()
#11 0x00000000005ff89a in ?? ()
#12 0x0000000000601c5e in mysql_execute_command() ()
#13 0x000000000060701c in mysql_parse() ()
#14 0x000000000060829a in dispatch_command() ()
#15 0x0000000000608b8a in do_command(THD*) ()
#16 0x00000000005fbd1d in handle_one_connection ()
#17 0x0000003be560686a in start_thread () from /lib64/libpthread.so.0
#18 0x0000003be4ede3bd in clone () from /lib64/libc.so.6
#19 0x0000000000000000 in ?? ()
```

堆栈需要自下而上来看。也就是说，线程当前正在执行的是 `pthread_cond_wait` 函数，这是由 `os_event_wait_low` 调用的。继续往下，看起来是线程试图进入到 InnoDB 内核（`srv_conc_enter_innodb`），但被放入了一个内部队列中（`os_event_wait_low`），原因应该是内核中的线程数已经超过 `innodb_thread_concurrency` 的限制。当然，要真正地发挥堆栈跟踪的价值需要将很多的信息聚合在一起来看。这种技术是由 Domas Mituzas 推广的，他以前是 MySQL 的支持工程师，开发了著名的穷人剖析器"poor man's profiler"。他目前在 Facebook 工作，和其他人一起开发了更多的收集和分析堆栈跟踪的工具。可以从他的这个网站发现更多的信息：*http://www.poormansprofiler.org*。

在 Percona Toolkit 中我们也开发了一个类似的穷人剖析器，叫做 *pt-pmp*。这是一个用 shell 和 *awk* 脚本编写的工具，可以将类似的堆栈跟踪输出合并到一起，然后通过 *sort|uniq|sort* 将最常见的条目在最前面输出。下面是一个堆栈跟踪的完整例子，通过此工具将重要的信息展示了出来。使用了 *-l5* 选项指定了堆栈跟踪不超过 5 层，以免因太 102
多前面部分相同而后面部分不同的跟踪信息而导致无法聚合到一起的情况，这样才能更好地显示到底在哪里产生了等待：

```
$ pt-pmp -l 5 stacktraces.txt
    507 pthread_cond_wait,one_thread_per_connection_end,handle_one_connection,
        start_thread,clone
    398 pthread_cond_wait,os_event_wait_low,srv_conc_enter_innodb,
        innodb_srv_conc_enter_innodb,ha_innodb::index_read
     83 pthread_cond_wait,os_event_wait_low,sync_array_wait_event,mutex_spin_wait,
        mutex_enter_func
     10 pthread_cond_wait,os_event_wait_low,os_aio_simulated_handle,fil_aio_wait,
        io_handler_thread
      7 pthread_cond_wait,os_event_wait_low,srv_conc_enter_innodb,
        innodb_srv_conc_enter_innodb,ha_innodb::general_fetch
      5 pthread_cond_wait,os_event_wait_low,sync_array_wait_event,rw_lock_s_lock_spin,
        rw_lock_s_lock_func
      1 sigwait,signal_hand,start_thread,clone,??
      1 select,os_thread_sleep,srv_lock_timeout_and_monitor_thread,start_thread,clone
      1 select,os_thread_sleep,srv_error_monitor_thread,start_thread,clone
      1 select,handle_connections_sockets,main
      1 read,vio_read_buff,::??,my_net_read,cli_safe_read


      1 pthread_cond_wait,os_event_wait_low,sync_array_wait_event,rw_lock_x_lock_low,
        rw_lock_x_lock_func
      1 pthread_cond_wait,MYSQL_BIN_LOG::wait_for_update,mysql_binlog_send,
        dispatch_command,do_command
      1 fsync,os_file_fsync,os_file_flush,fil_flush,log_write_up_to
```

第一行是 MySQL 中非常典型的空闲线程的一种特征，所以可以忽略。第二行才是最有意思的地方，看起来大量的线程正在准备进入到 InnoDB 内核中，但都被阻塞了。从第三行则可以看到许多线程都在等待某些互斥锁，但具体的是什么锁不清楚，因为堆栈跟踪更深的层次被截断了。如果需要确切地知道是什么互斥锁，则需要使用更大的 *-l* 选项重跑一次。一般来说，这个堆栈跟踪显示很多线程都在等待进入到 InnoDB，这是为什么呢？这个工具并不清楚，需要从其他的地方来入手。

从前面的堆栈跟踪和 *oprofile* 报表来看，如果不是 MySQL 和 InnoDB 源码方面的专家，这种类型的分析很难进行。如果用户在进行此类分析时碰到问题，通常需要求助于这样的专家才行。

在下面的例子中，通过剖析和等待分析都无法发现服务器的问题，需要使用另外一种不同的诊断技术。

3.4.3 一个诊断案例

在本节中，我们将逐步演示一个客户实际碰到的间歇性性能问题的诊断过程。这个案例的诊断需要具备 MySQL、InnoDB 和 GNU/Linux 的相关知识。但这不是我们要讨论的
103 重点。要尝试从疯狂中找到条理：阅读本节并保持对之前的假设和猜测的关注，保持对之前基于合理性和基于可度量的方式的关注，等等。我们在这里深入研究一个具体和详

细的案例，为的是找到一个简单的一般性的方法。

在尝试解决其他人提出的问题之前，先要明确两件事情，并且最好能够记录下来，以免遗漏或者遗忘：

1. 首先，问题是什么？一定要清晰地描述出来，费力去解决一个错误的问题是常有的事。在这个案例中，用户抱怨说每隔一两天，服务器就会拒绝连接，报 max_connections 错误。这种情况一般会持续几秒到几分钟，发生的时间非常随机。
2. 其次，为解决问题已经做过什么操作？在这个案例中，用户没有为这个问题做过任何操作。这个信息非常有帮助，因为很少有其他事情会像另外一个人来描述一件事情发生的确切顺序和曾做过的改变及其后果一样难以理解（尤其是他们还是在经过几个不眠之夜后满嘴咖啡味道地在电话里绝望呐喊的时候）。如果一台服务器遭受过未知的变更，产生了未知的结果，问题就更难解决了，尤其是时间又非常有限的时候。

搞清楚这两个问题后，就可以开始了。不仅需要去了解服务器的行为，也需要花点时间去梳理一下服务器的状态、参数配置，以及软硬件环境。使用 *pt-summary* 和 *pt-mysql-summary* 工具可以获得这些信息。简单地说，这个例子中的服务器有 16 个 CPU 核心，12GB 内存，数据量有 900MB，且全部采用 InnoDB 引擎，存储在一块 SSD 固态硬盘上。服务器的操作系统是 GNU/Linux、MySQL 版本 5.1.37，使用的存储引擎版本是 InnoDB plugin 1.0.4。之前我们已经为这个客户解决过一些异常问题，所以对其系统已经比较了解。过去数据库从来没有出过问题，大多数问题都是由于应用程序的不良行为导致的。初步检查了服务器也没有发现明显的问题。查询有一些优化的空间，但大多数情况下响应时间都不到 10 毫秒。所以我们认为正常情况下数据库服务器运行良好（这一点比较重要，因为很多问题一开始只是零星地出现，慢慢地累积成大问题。比如 RAID 阵列中坏了一块硬盘这种情况）。

> 这个案例研究可能有点乏味。这里我们不厌其烦地展示所有的诊断数据，解释所有的细节，对几个不同的可能性深入进去追查原因。在实际工作中，其实不会对每个问题都采用这样缓慢而冗长的方式，也不推荐大家这样做。这里只是为了更好地演示案例而已。

我们安装好诊断工具，在 Threads_connected 上设置触发条件，正常情况下 Threads_connected 的值一般都少于 15，但在发生问题时该值可能飙升到几百。下面我们会先给出一个样本数据的收集结果，后续再来评论。首先试试看，你能否从大量的输出中找出 104
问题的重点在哪里：

- 查询活动从 1 000 到 10 000 的 QPS，其中有很多是“垃圾”命令，比如 ping 一下服务器确认其是否存活。其余的大部分是 SELECT 命令，大约每秒 300 ~ 2 000 次，只有很少的 UPDATE 命令（大约每秒五次）。
- 在 SHOW PROCESSLIST 中主要有两种类型的查询，只是在 WHERE 条件中的值不一样。下面是查询状态的汇总数据：

```
$ grep State: processlist.txt | sort | uniq -c | sort -rn
    161  State: Copying to tmp table
    156  State: Sorting result
    136  State: statistics
     50  State: Sending data
     24  State: NULL
     13  State:
      7  State: freeing items
      7  State: cleaning up
      1  State: storing result in query cache
      1  State: end
```

- 大部分查询都是索引扫描或者范围扫描，很少有全表扫描或者表关联的情况。
- 每秒大约有 20 ~ 100 次排序，需要排序的行大约有 1 000 到 12 000 行。
- 每秒大约创建 12 ~ 90 个临时表，其中有 3 ~ 5 个是磁盘临时表。
- 没有表锁或者查询缓存的问题。
- 在 SHOW INNODB STATUS 中可以观察到主要的线程状态是“flushing buffer pool pages”，但只有很少的脏页需要刷新（Innodb_buffer_pool_pages_dirty），Innodb_buffer_pool_pages_flushed 也没有太大的变化，日志顺序号（log sequence number）和最后检查点（last checkpoint）之间的差距也很少。InnoDB 缓存池也还远没有用满；缓存池比数据集还要大很多。大多数线程在等待 InnoDB 队列：“12 queries inside InnoDB, 495 queries in queue”（12 个查询在 InnoDB 内部执行，495 个查询在队列中）。
- 每秒捕获一次 *iostat* 输出，持续 30 秒。从输出可以发现没有磁盘读，而写操作则接近了“天花板”，所以 I/O 平均等待时间和队列长度都非常高。下面是部分输出结果，为便于打印输出，这里截取了部分字段：

```
  r/s    w/s rsec/s    wsec/s avgqu-sz  await svctm  %util
 1.00 500.00   8.00  86216.00     5.05  11.95  0.59  29.40
 0.00 451.00   0.00 206248.00   123.25 238.00  1.90  85.90
 0.00 565.00   0.00 269792.00   143.80 245.43  1.77 100.00
 0.00 649.00   0.00 309248.00   143.01 231.30  1.54 100.10
 0.00 589.00   0.00 281784.00   142.58 232.15  1.70 100.00
 0.00 384.00   0.00 162008.00    71.80 238.39  1.73  66.60
 0.00  14.00   0.00    400.00     0.01   0.93  0.36   0.50
 0.00  13.00   0.00    248.00     0.01   0.92  0.23   0.30
 0.00  13.00   0.00    408.00     0.01   0.92  0.23   0.30
```

105

- *vmstat* 的输出也验证了 *iostat* 的结果，并且 CPU 的大部分时间是空闲的，只是偶尔在写尖峰时有一些 I/O 等待时间（最高约占 9% 的 CPU）。

是不是感觉脑袋里塞满了东西？当你深入一个系统的细节并且没有任何先入为主（或者故意忽略了）的观念时，很容易碰到这种情况，最终只能检查所有可能的情况。很多被检查的地方最终要么是完全正常的，要么发现是问题导致的结果而不是问题产生的原因。尽管此时我们会有很多关于问题原因的猜测，但还是需要继续检查下面给出的 *oprofile* 报表，并且在给出更多数据的时候添加一些评论和解释：

```
samples  %        image name       app name         symbol name
 473653  63.5323  no-vmlinux       no-vmlinux       /no-vmlinux
  95164  12.7646  mysqld           mysqld           /usr/libexec/mysqld
  53107   7.1234  libc-2.10.1.so   libc-2.10.1.so   memcpy
  13698   1.8373  ha_innodb.so     ha_innodb.so     build_template()
  13059   1.7516  ha_innodb.so     ha_innodb.so     btr_search_guess_on_hash
  11724   1.5726  ha_innodb.so     ha_innodb.so     row_sel_store_mysql_rec
   8872   1.1900  ha_innodb.so     ha_innodb.so     rec_init_offsets_comp_ordinary
   7577   1.0163  ha_innodb.so     ha_innodb.so     row_search_for_mysql
   6030   0.8088  ha_innodb.so     ha_innodb.so     rec_get_offsets_func
   5268   0.7066  ha_innodb.so     ha_innodb.so     cmp_dtuple_rec_with_match
```

这里大多数符号（symbol）代表的意义并不是那么明显，而大部分的时间都消耗在内核符号（no-vmlinux）[注17] 和一个通用的 *mysqld* 符号中，这两个符号无法告诉我们更多的细节[注18]。不要被多个 ha_innodb.so 符号分散了注意力，看一下它们占用的百分比就知道了，不管它们在做什么，其占用的时间都很少，所以应该不会是问题所在。这个例子说明，仅仅从剖析报表出发是无法得到解决问题的结果的。我们追踪的数据是错误的。如果遇到上述例子这样的情况，需要继续检查其他的数据，寻找问题根源更明显的证据。

到这里，如果希望从 *gdb* 的堆栈跟踪进行等待分析，请参考 3.4.2 节的最后部分内容。那个案例就是我们当前正在诊断的这个问题。回想一下，当时的堆栈跟踪分析的结果是正在等待进入到 InnoDB 内核，所以 SHOW INNODB STATUS 的输出结果中有“12 queries inside InnoDB, 495 queries in queue”。

从上面的分析发现问题的关键点了吗？没有。我们看到了许多不同问题可能的症状，根据经验和直觉可以推测至少有两个可能的原因。但也有一些没有意义的地方。如果再次 106
检查一下 *iostat* 的输出，可以发现 wsec/s 列显示了至少在 6 秒内，服务器每秒写入了几百 MB 的数据到磁盘。每个磁盘扇区是 512B，所以这里采样的结果显示每秒最多写入了 150MB 数据。然而整个数据库也只有 900MB 大小，系统的压力又主要是 SELECT

注 17： 理论上，我们需要内核符号（kernel symbol）才能理解内核中发生了什么。实际上，安装内核符号可能会比较麻烦，并且从 *vmstat* 的输出可以看到系统 CPU 的利用率很低，所以即使安装了，很可能也会发现内核大多数是处于“sleeping”（睡眠）状态的。

注 18： 这看起来是一个编译有问题的 MySQL 版本。

查询。怎么会出现这样的情况呢？

对一个系统进行检查的时候，应该先问一下自己，是否也碰到过上面这种明显不合理的问题，如果有就需要深入调查。应该尽量跟进每一个可能的问题直到发现结果，而不要被离题太多的各种情况分散了注意力，以致最后都忘记了最初要调查的问题。可以把问题写在小纸条上，检查一个划掉一个，最后再确认一遍所有的问题都已经完成调查[注19]。

在这一点上，我们可以直接得到一个结论，但却可能是错误的。可以看到主线程的状态是 InnoDB 正在刷新脏页。在状态输出中出现这样的情况，一般都意味着刷新已经延迟了。我们知道这个版本的 InnoDB 存在“疯狂刷新”的问题（或者也被称为检查点停顿）。发生这样的情况是因为 InnoDB 没有按时间均匀分布刷新请求，而是隔一段时间突然请求一次强制检查点导致大量刷新操作。这种机制可能会导致 InnoDB 内部发生严重的阻塞，导致所有的操作需要排队等待进入内核，从而引发 InnoDB 上一层的服务器产生堆积。在第 2 章中演示的例子就是一个因为“疯狂刷新”而导致性能周期性下跌的问题。很多类似的问题都是由于强制检查点导致的，但在这个案例中却不是这个问题。有很多方法可以证明，最简单的方法是查看 SHOW STATUS 的计数器，追踪一下 Innodb_buffer_pool_pages_flushed 的变化，之前已经提到了，这个值并没有怎么增加。另外，注意到 InnoDB 缓冲池中也没有大量的脏页需要刷新，肯定不到几百 MB。这并不值得惊讶，因为这个服务器的工作压力几乎都是 SELECT 查询。所以可以得到一个初步的结论，我们要关注的不是 InnoDB 刷新的问题，而应该是刷新延迟的问题，但这只是一个现象，而不是原因。根本的原因是磁盘的 I/O 已经饱和，InnoDB 无法完成其 I/O 操作。至此我们消除了一个可能的原因，可以从基于直觉的原因列表中将其划掉了。

从结果中将原因区别出来有时候会很困难。当一个问题看起来很眼熟的时候，也可以跳过调查阶段直接诊断。当然最好不要走这样的捷径，但有时候依靠直觉也非常重要。如果有什么地方看起来很眼熟，明智的做法还是需要花一点时间去测量一下其充分必要条件，以证明其是否就是问题所在。这样可以节省大量时间，避免查看大量其他的系统和
107 性能数据。不过也不要过于相信直觉而直接下结论，不要说“我之前见过这样的问题，肯定就是同样的问题”。而是应该去收集相关的证据，尤其是能证明直觉的证据。

下一步是尝试找出是什么导致了服务器的 I/O 利用率异常的高。首先应该注意到前面已经提到过的“服务器有连续几秒内每秒写入了几百 MB 数据到磁盘，而数据库一共只有 900MB 大小，怎么会发生这样的情况？”，注意到这里已经隐式地假设是数据库导致了磁盘写入。那么有什么证据表明是数据库导致的呢？当你有未经证实的想法，或者觉得不可思议时，如果可能的话应该去进行测量，然后排除掉一些怀疑。

注 19： 或者换个说法，不要把所有的鸡蛋都混在一个篮子里。

我们看到了两种可能性：要么是数据库导致了 I/O（如果能找到源头的话，那么可能就找到了问题的原因）；要么不是数据库导致了所有的 I/O 而是其他什么导致的，而系统因为缺少 I/O 资源影响了数据库性能。我们也很小心地尽力避免引入另外一个隐式的假设：磁盘很忙并不一定意味着 MySQL 会有问题。要记住，这个服务器主要的压力是内存读取，所以也很可能出现磁盘长时间无法响应但没有造成严重问题的现象。

如果你一直跟随我们的推理逻辑，就可以发现还需要回头检查一下另外一个假设。我们已经知道磁盘设备很忙，因为其等待时间很高。对于固态硬盘来说，其 I/O 平均等待时间一般不会超过 1/4 秒。实际上，从 *iostat* 的输出结果也可以发现磁盘本身的响应还是很快的，但请求在块设备队列中等待很长的时间才能进入到磁盘设备。但要记住，这只是 *iostat* 的输出结果，也可能是错误的信息。

究竟是什么导致了性能低下？

当一个资源变得效率低下时，应该了解一下为什么会这样。有如下可能的原因：

1. 资源被过度使用，余量已经不足以正常工作。
2. 资源没有被正确配置。
3. 资源已经损坏或者失灵。

回到上面的例子中，*iostat* 的输出显示可能是磁盘的工作负载太大，也可能是配置不正确（在磁盘响应很快的情况下，为什么 I/O 请求需要排队这么长时间才能进入到磁盘？）。然而，比较系统的需求和现有容量对于确定问题在哪里是很重要的一部分。大量的基准测试证明这个客户使用的这种 SSD 是无法支撑几百 MB/s 的写操作的。所以，尽管 *iostat* 的结果表明磁盘的响应是正常的，也不一定是完全正确的。在这个案例中，我们没有办法证明磁盘的响应比 *iostat* 的结果中所说的要慢，但这种情况还是有可能的。所以这不能改变我们的看法：可能是磁盘被滥用[注20]，或者是错误的配置，或者两者兼而有之，是性能低下的罪魁祸首。

在检查过所有诊断数据之后，接下来的任务就很明显了：测量出什么导致了 I/O 消耗。不幸的是，客户当前使用的 GNU/Linux 版本对此的支持不力。通过一些工作我们可以做一些相对准确的猜测，但首先还是需要探索一下其他的可能性。我们可以测量有多少 I/O 来自 MySQL，但客户使用的 MySQL 版本较低以致缺乏一些诊断功能，所以也无法提供确切有利的支持。 108

注 20： 也有人会拨打 1-800 热线电话。

作为替代，基于我们已经知道 MySQL 如何使用磁盘，我们来观察 MySQL 的 I/O 情况。通常来说，MySQL 只会写数据、日志、排序文件和临时表到磁盘。从前面的状态计数器和其他信息来看，首先可以排除数据和日志的写入问题。那么，只能假设 MySQL 突然写入大量数据到临时表或者排序文件，如何来观察这种情况呢？有两个简单的方法：一是观察磁盘的可用空间，二是通过 *lsof* 命令观察服务器打开的文件句柄。这两个方法我们都采用了，结果也足以满足我们的需求。下面是问题期间每秒运行 *df–h* 的结果：

```
Filesystem  Size  Used  Avail  Use%  Mounted  on
/dev/sda3   58G   20G   36G    36%   /
/dev/sda3   58G   20G   36G    36%   /
/dev/sda3   58G   19G   36G    35%   /
/dev/sda3   58G   19G   36G    35%   /
/dev/sda3   58G   19G   36G    35%   /
/dev/sda3   58G   19G   36G    35%   /
/dev/sda3   58G   18G   37G    33%   /
/dev/sda3   58G   18G   37G    33%   /
/dev/sda3   58G   18G   37G    33%   /
```

下面则是 *lsof* 的数据，因为某些原因我们每五秒才收集一次。我们简单地将 *mysqld* 在 */tmp* 中打开的文件大小做了加总，并且把总大小和采样时的时间戳一起输出到结果文件中：

```
$ awk '
  /mysqld.*tmp/ {
    total += $7;
  }
  /^Sun Mar 28/ && total {
    printf "%s %7.2f MB\n", $4, total/1024/1024;
    total = 0;
  }' lsof.txt
18:34:38 1655.21 MB
18:34:43    1.88 MB
18:34:48    1.88 MB
18:34:53    1.88 MB
18:34:58    1.88 MB
```

从这个数据可以看出，在问题之初 MySQL 大约写了 1.5GB 的数据到临时表，这和之前在 SHOW PROCESSLIST 中有大量的"Copying to tmp table"相吻合。这个证据表明可能是
109 某些效率低下的查询风暴耗尽了磁盘资源。根据我们的工作直觉，出现这种情况比较普遍的一个原因是缓存失效。当 *memcached* 中所有缓存的条目同时失效，而又有很多应用需要同时访问的时候，就会出现这种情况。我们给开发人员出示了部分采样到的查询，并讨论这些查询的作用。实际情况是，缓存同时失效就是罪魁祸首（这验证了我们的直觉）。一方面开发人员在应用层面解决缓存失效的问题；另一方面我们也修改了查询，避免使用磁盘临时表。这两个方法的任何一个都可以解决问题，当然最好是两个都实施。

如果读者一直顺着我们前面的思路读下来，可能还会有一些疑问。在这里我们可以稍微解释一下（我们在本章引用的方法在审阅的时候已经检查过一遍）：

为什么我们不一开始就优化慢查询？

因为问题不在于慢查询，而是“太多连接”的错误。当然，因为慢查询，太多查询的时间过长而导致连接堆积在逻辑上也是成立的。但也有可能是其他原因导致连接过多。如果没有找到问题的真正原因，那么回头查看慢查询或其他可能的原因，看是否能够改善是很自然的事情[注 21]。但这样做大多时候会让问题变得更糟。如果你把一辆车开到机械师那里抱怨说有异响，假如机械师没有指出异响的原因，也不去检查其他的地方，而是直接做了四轮平衡和更换变速箱油，然后把账单扔给你，你也会觉得不爽的吧？

但是查询由于糟糕的执行计划而执行缓慢不是一种警告吗？

在事故中确实如此。但慢查询到底是原因还是结果？在深入调查前是无法知晓的。记住，在正常的时候这个查询也是正常运行的。一个查询需要执行 filesort 和创建临时表并不一定意味着就是有问题的。尽管消除 filesort 和临时表通常来说是“最佳实践”。

通常的“最佳实践”自然有它的道理，但不一定是解决某些特殊问题的“灵丹妙药”。比如说问题可能是因为很简单的配置错误。我们碰到过很多这样的案例，问题本来是由于错误的配置导致的，却去优化查询，这不但浪费了时间，也使得真正问题被解决的时间被拖延了。

如果缓存项被重新生成了很多次，是不是会导致产生很多同样的查询呢？

这个问题我们确实还没有调查到。如果是多线程重新生成同样的缓存项，那么确实有可能导致产生很多同样的查询（这和很多同类型的查询不同，比如 WHERE 子句中 110
的参数可能不一样）。注意到这样会刺激我们的直觉，并更快地带我们找到问题的解决方案。

每秒有几百次 SELECT 查询，但只有五次 UPDATE。怎么能确定这五次 UPDATE 的压力不会导致问题呢？

这些 UPDATE 有可能对服务器造成很大的压力。我们没有将真正的查询语句展示出来，因为这样可能会将事情搞得更杂乱。但有一点很明确，某种查询的绝对数量不一定有意义。

I/O 风暴最初的证据看起来不是很充分？

是的，确实是这样。有很多种解释可以说明为什么一个这么小的数据库可以产生这么大量的写入磁盘，或者说为什么磁盘的可用空间下降得这么快。这个问题中使用的 MySQL 和 GNU/Linux 版本都很难对一些东西进行测量（但不是说完全不可能）。

注 21：就像常说的“当你手中有了锤子，所有的东西看起来都是钉子”一样。

尽管在很多时候我们可能扮演“魔鬼代言人”的角色，但我们还是以尽量平衡成本和潜在的利益为第一优先级。越是难以准确测量的时候，成本 / 收益比越攀升，我们也更愿意接受不确定性。

之前说过“数据库过去从来没出过问题”是一种偏见吗？

是的，这就是偏见。如果抓住问题，很好；如果没有，也可以是证明我们都有偏见的很好例子。

至此我们要结束这个案例的学习了。需要指出的是，如果使用了诸如 New Relic 这样的剖析工具，即使没有我们的参与，也可能解决这个问题。

3.5 其他剖析工具

我们已经演示了很多剖析 MySQL、操作系统及查询的方法。我们也演示了那些我们觉得很有用的案例。当然，通过本书，我们还会展示更多工具和技术来检查和测量系统。但是等一下，本章还有更多工具没介绍呢。

3.5.1 使用 USER_STATISTICS 表

Percona Server 和 MariaDB 都引入了一些额外的对象级别使用统计的 INFORMATION_SCHEMA 表，这些最初是由 Google 开发的。这些表对于查找服务器各部分的实际使用情况非常有帮助。在一个大型企业中，DBA 负责管理数据库，但其对开发缺少话语权，那么通过这些表就可以对数据库活动进行测量和审计，并且强制执行使用策略。对于像共享主机环境这样的多租户环境也同样有用。另外，在查找性能问题时，这些表也可以帮助找出数据库中什么地方花费了最多的时间，或者什么表或索引使用得最频繁，抑或最不频繁。下面就是这些表：

111

```
mysql> SHOW TABLES FROM INFORMATION_SCHEMA LIKE '%_STATISTICS';
+---------------------------------------------+
| Tables_in_information_schema (%_STATISTICS) |
+---------------------------------------------+
| CLIENT_STATISTICS                           |
| INDEX_STATISTICS                            |
| TABLE_STATISTICS                            |
| THREAD_STATISTICS                           |
| USER_STATISTICS                             |
+---------------------------------------------+
```

这里我们不会详细地演示针对这些表的所有有用的查询，但有几个要点要说明一下：

- 可以查找使用得最多或者使用得最少的表和索引，通过读取次数或者更新次数，或者两者一起排序。

- 可以查找出从未使用的索引，可以考虑删除之。
- 可以看看复制用户的 CONNECTED_TIME 和 BUSY_TIME，以确认复制是否会很难跟上主库的进度。

在 MySQL 5.6 中，Performance Schema 中也添加了很多类似上面这些功能的表。

3.5.2 使用 strace

strace 工具可以调查系统调用的情况。有好几种可以使用的方法，其中一种是计算系统调用的时间并打印出来：

```
$ strace -cfp $(pidof mysqld)
Process 12648 attached with 17 threads - interrupt to quit
^CProcess 12648 detached
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 73.51    0.608908       13839        44           select
 24.38    0.201969       20197        10           futex
  0.76    0.006313           1     11233         3 read
  0.60    0.004999         625         8           unlink
  0.48    0.003969          22       180           write
  0.23    0.001870          11       178           pread64
  0.04    0.000304           0      5538           _llseek
[some lines omitted for brevity]
------ ----------- ----------- --------- --------- ----------------
100.00    0.828375                 17834        46 total
```

这种用法和 *oprofile* 有点像。但是 *oprofile* 还可以剖析程序的内部符号，而不仅仅是系统调用。另外，*strace* 拦截系统调用使用的是不同于 *oprofile* 的技术，这会有一些不可预期性，开销也更大些。*strace* 度量时使用的是实际时间，而 *oprofile* 使用的是花费的 CPU 周期。举个例子，当 I/O 等待出现问题的时候，*strace* 能将它们显示出来，因为它从诸如 read 或者 pread64 这样的系统调用开始计时，直到调用结束。但 *oprofile* 不会这样，因为 I/O 系统调用并不会真正地消耗 CPU 周期，而只是等待 I/O 完成而已。 112

我们会在需要的时候使用 *oprofile*，因为 *strace* 对像 *mysqld* 这样有大量线程的场景会产生一些副作用。当 *strace* 附加上去后，*mysqld* 的运行会变得很慢，因此不适合在产品环境中使用。但在某些场景中 *strace* 还是相当有用的，Percona Toolkit 中有一个叫做 *pt-ioprofile* 的工具就是使用 *strace* 来生成 I/O 活动的剖析报告的。这个工具很有帮助，可以证明或者驳斥某些难以测量的情况下的一些观点，此时其他方法很难达到目的（如果运行的是 MySQL 5.6，使用 Performance Schema 也可以达到目的）。

3.6 总结

本章给出了一些基本的思路和技术，有助于你成功地进行性能优化。正确的思维方式是开启系统的全部潜力和应用本书其他章节提供的知识的关键。下面是我们试图演示的一些基本知识点：

- 我们认为定义性能最有效的方法是响应时间。
- 如果无法测量就无法有效地优化，所以性能优化工作需要基于高质量、全方位及完整的响应时间测量。
- 测量的最佳开始点是应用程序，而不是数据库。即使问题出在底层的数据库，借助良好的测量也可以很容易地发现问题。
- 大多数系统无法完整地测量，测量有时候也会有错误的结果。但也可以想办法绕过一些限制，并得到好的结果（但是要能意识到所使用的方法的缺陷和不确定性在哪里）。
- 完整的测量会产生大量需要分析的数据，所以需要用到剖析器。这是最佳的工具，可以帮助将重要的问题冒泡到前面，这样就可以决定从哪里开始分析会比较好。
- 剖析报告是一种汇总信息，掩盖和丢弃了太多细节。而且它不会告诉你缺少了什么，所以完全依赖剖析报告也是不明智的。
- 有两种消耗时间的操作：工作或者等待。大多数剖析器只能测量因为工作而消耗的时间，所以等待分析有时候是很有用的补充，尤其是当 CPU 利用率很低但工作却一直无法完成的时候。
- 优化和提升是两回事。当继续提升的成本超过收益的时候，应当停止优化。
- 113 注意你的直觉，但应该只根据直觉来指导解决问题的思路，而不是用于确定系统的问题。决策应当尽量基于数据而不是感觉。

总体来说，我们认为解决性能问题的方法，首先是要澄清问题，然后选择合适的技术来解答这些问题。如果你想尝试提升服务器的总体性能，那么一个比较好的起点是将所有查询记录到日志中，然后利用 *pt-query-digest* 工具生成系统级别的剖析报告。如果是要追查某些性能低下的查询，记录和剖析的方法也会有帮助。可以把精力放在寻找那些消耗时间最多的、导致了糟糕的用户体验的，或者那些高度变化的，抑或有奇怪的响应时间直方图的查询。当找到了这些“坏”查询时，要钻取 *pt-query-digest* 报告中包含的该查询的详细信息，或者使用 `SHOW PROFILE` 及其他诸如 `EXPLAIN` 这样的工具。

如果找不到这些查询性能低下的原因，那么也可能是遇到了服务器级别的性能问题。这时，可以较高精度测量和绘制服务器状态计数器的细节信息。如果通过这样的分析重现了问题，则应该通过同样的数据制定一个可靠的触发条件，来收集更多的诊断数据。多花费一点时间来确定可靠的触发条件，尽量避免漏检或者误报。如果已经可以捕获故障

活动期间的数据，但还是无法找到其根本原因，则要么尝试捕获更多的数据，要么尝试寻求帮助。

我们无法完整地测量工作系统，但说到底它们都是某种状态机，所以只要足够细心，逻辑清晰并且坚持下去，通常来说都能得到想要的结果。要注意的是不要把原因和结果搞混了，而且在确认问题之前也不要随便针对系统做变动。

理论上纯粹的自顶向下的方法分析和详尽的测量只是理想的情况，而我们常常需要处理的是真实系统。真实系统是复杂且无法充分测量的，所以我们只能根据情况尽力而为。使用诸如 *pt-query-digest* 和 MySQL 企业监控器的查询分析器这样的工具并不完美，通常都不会给出问题根源的直接证据。但真的掌握了以后，已经足以完成大部分的优化诊断工作了。

第 4 章 Schema与数据类型优化

良好的逻辑设计和物理设计是高性能的基石，应该根据系统将要执行的查询语句来设计schema，这往往需要权衡各种因素。例如，反范式的设计可以加快某些类型的查询，但同时可能使另一些类型的查询变慢。比如添加计数表和汇总表是一种很好的优化查询的方式，但这些表的维护成本可能会很高。MySQL 独有的特性和实现细节对性能的影响也很大。

本章和聚焦在索引优化的下一章，覆盖了 MySQL 特有的 schema 设计方面的主题。我们假设读者已经知道如何设计数据库，所以本章既不会介绍如何入门数据库设计，也不会讲解数据库设计方面的深入内容。这一章关注的是 MySQL 数据库的设计，主要介绍的是 MySQL 数据库设计与其他关系型数据库管理系统的区别。如果需要学习数据库设计方面的基础知识，建议阅读 Clare Churcher 的 *Beginning Database Design*（Apress 出版社）一书。

本章内容是为接下来的两个章节做铺垫。在这三章中，我们将讨论逻辑设计、物理设计和查询执行，以及它们之间的相互作用。这既需要关注全局，也需要专注细节。还需要理解整个系统以便弄清楚各个部分如何相互影响。如果在阅读完索引和查询优化章节后再回头来看这一章，也许会发现本章很有用，很多讨论的议题不能孤立地考虑。

4.1 选择优化的数据类型

MySQL 支持的数据类型非常多，选择正确的数据类型对于获得高性能至关重要。不管存储哪种类型的数据，下面几个简单的原则都有助于做出更好的选择。

更小的通常更好。

一般情况下，应该尽量使用可以正确存储数据的最小数据类型[注1]。更小的数据类型通

注 1： 例如只需要存 0~200，tinyint unsigned 更好。——译者注

116 常更快，因为它们占用更少的磁盘、内存和CPU缓存，并且处理时需要的CPU周期也更少。

但是要确保没有低估需要存储的值的范围，因为在schema中的多个地方增加数据类型的范围是一个非常耗时和痛苦的操作。如果无法确定哪个数据类型是最好的，就选择你认为不会超过范围的最小类型。（如果系统不是很忙或者存储的数据量不多，或者是在可以轻易修改设计的早期阶段，那之后修改数据类型也比较容易）。

简单就好

简单数据类型的操作通常需要更少的CPU周期。例如，整型比字符操作代价更低，因为字符集和校对规则（排序规则）使字符比较比整型比较更复杂。这里有两个例子：一个是应该使用MySQL内建的类型[注2]而不是字符串来存储日期和时间，另外一个是应该用整型存储IP地址。稍后我们将专门讨论这个话题。

尽量避免 NULL

很多表都包含可为NULL（空值）的列，即使应用程序并不需要保存NULL也是如此，这是因为可为NULL是列的默认属性[注3]。通常情况下最好指定列为NOT NULL，除非真的需要存储NULL值。

如果查询中包含可为NULL的列，对MySQL来说更难优化，因为可为NULL的列使得索引、索引统计和值比较都更复杂。可为NULL的列会使用更多的存储空间，在MySQL里也需要特殊处理。当可为NULL的列被索引时，每个索引记录需要一个额外的字节，在MyISAM里甚至还可能导致固定大小的索引（例如只有一个整数列的索引）变成可变大小的索引。

通常把可为NULL的列改为NOT NULL带来的性能提升比较小，所以（调优时）没有必要首先在现有schema中查找并修改掉这种情况，除非确定这会导致问题。但是，如果计划在列上建索引，就应该尽量避免设计成可为NULL的列。

当然也有例外，例如值得一提的是，InnoDB使用单独的位（bit）存储NULL值，所以对于稀疏数据[注4]有很好的空间效率。但这一点不适用于MyISAM。

在为列选择数据类型时，第一步需要确定合适的大类型：数字、字符串、时间等。这通常是很简单的，但是我们会提到一些特殊的不是那么直观的案例。

下一步是选择具体类型。很多MySQL的数据类型可以存储相同类型的数据，只是存储的长度和范围不一样、允许的精度不同，或者需要的物理空间（磁盘和内存空间）不同。相同大类型的不同子类型数据有时也有一些特殊的行为和属性。

117 例如，DATETIME和TIMESAMP列都可以存储相同类型的数据：时间和日期，精确到秒。

注2： date, time, datatime——译者注

注3： 如果定义表结构时没有指定列为NOT NULL，默认都是允许为NULL的。

注4： 很多值为NULL，只有少数行的列有非NULL值。——译者注

然而 TIMESTAMP 只使用 DATETIME 一半的存储空间，并且会根据时区变化，具有特殊的自动更新能力。另一方面，TIMESTAMP 允许的时间范围要小得多，有时候它的特殊能力会成为障碍。

本章只讨论基本的数据类型。MySQL 为了兼容性支持很多别名，例如 INTEGER、BOOL，以及 NUMERIC。它们都只是别名。这些别名可能令人不解，但不会影响性能。如果建表时采用数据类型的别名，然后用 SHOW CREATE TABLE 检查，会发现 MySQL 报告的是基本类型，而不是别名。

4.1.1 整数类型

有两种类型的数字：整数（whole number）和实数（real number）。如果存储整数，可以使用这几种整数类型：TINYINT，SMALLINT，MEDIUMINT，INT，BIGINT。分别使用 8，16，24，32，64 位存储空间。它们可以存储的值的范围从 $-2^{(N-1)}$ 到 $2^{(N-1)}-1$，其中 N 是存储空间的位数。

整数类型有可选的 UNSIGNED 属性，表示不允许负值，这大致可以使正数的上限提高一倍。例如 TINYINT. UNSIGNED 可以存储的范围是 0 ～ 255，而 TINYINT 的存储范围是 −128 ～ 127。

有符号和无符号类型使用相同的存储空间，并具有相同的性能，因此可以根据实际情况选择合适的类型。

你的选择决定 MySQL 是怎么在内存和磁盘中保存数据的。然而，整数计算一般使用 64 位的 BIGINT 整数，即使在 32 位环境也是如此。（一些聚合函数是例外，它们使用 DECIMAL 或 DOUBLE 进行计算）。

MySQL 可以为整数类型指定宽度，例如 INT(11)，对大多数应用这是没有意义的：它不会限制值的合法范围，只是规定了 MySQL 的一些交互工具（例如 MySQL 命令行客户端）用来显示字符的个数。对于存储和计算来说，INT(1) 和 INT(20) 是相同的。

> 一些第三方存储引擎，比如 Infobright，有时也有自定义的存储格式和压缩方案，并不一定使用常见的 MySQL 内置引擎的方式。

4.1.2 实数类型 118

实数是带有小数部分的数字。然而，它们不只是为了存储小数部分；也可以使用 DECIMAL 存储比 BIGINT 还大的整数。MySQL 既支持精确类型，也支持不精确类型。

FLOAT 和 DOUBLE 类型支持使用标准的浮点运算进行近似计算。如果需要知道浮点运算是怎么计算的，则需要研究所使用的平台的浮点数的具体实现。

DECIMAL 类型用于存储精确的小数。在 MySQL 5.0 和更高版本，DECIMAL 类型支持精确计算。MySQL 4.1 以及更早版本则使用浮点运算来实现 DECIAML 的计算，这样做会因为精度损失导致一些奇怪的结果。在这些版本的 MySQL 中，DECIMAL 只是一个“存储类型”。

因为 CPU 不支持对 DECIMAL 的直接计算，所以在 MySQL 5.0 以及更高版本中，MySQL 服务器自身实现了 DECIMAL 的高精度计算。相对而言，CPU 直接支持原生浮点计算，所以浮点运算明显更快。

浮点和 DECIMAL 类型都可以指定精度。对于 DECIMAL 列，可以指定小数点前后所允许的最大位数。这会影响列的空间消耗。MySQL 5.0 和更高版本将数字打包保存到一个二进制字符串中（每 4 个字节存 9 个数字）。例如，DECIMAL(18,9) 小数点两边将各存储 9 个数字，一共使用 9 个字节：小数点前的数字用 4 个字节，小数点后的数字用 4 个字节，小数点本身占 1 个字节。

MySQL 5.0 和更高版本中的 DECIMAL 类型允许最多 65 个数字。而早期的 MySQL 版本中这个限制是 254 个数字，并且保存为未压缩的字符串（每个数字一个字节）。然而，这些（早期）版本实际上并不能在计算中使用这么大的数字，因为 DECIMAL 只是一种存储格式；在计算中 DECIMAL 会转换为 DOUBLE 类型。

有多种方法可以指定浮点列所需要的精度，这会使得 MySQL 悄悄选择不同的数据类型，或者在存储时对值进行取舍。这些精度定义是非标准的，所以我们建议只指定数据类型，不指定精度。

浮点类型在存储同样范围的值时，通常比 DECIMAL 使用更少的空间。FLOAT 使用 4 个字节存储。DOUBLE 占用 8 个字节，相比 FLOAT 有更高的精度和更大的范围。和整数类型一样，能选择的只是存储类型；MySQL 使用 DOUBLE 作为内部浮点计算的类型。

因为需要额外的空间和计算开销，所以应该尽量只在对小数进行精确计算时才使用 DECIMAL——例如存储财务数据。但在数据量比较大的时候，可以考虑使用 BIGINT 代替
119 DECIMAL，将需要存储的货币单位根据小数的位数乘以相应的倍数即可。假设要存储财务数据精确到万分之一分，则可以把所有金额乘以一百万，然后将结果存储在 BIGINT 里，这样可以同时避免浮点存储计算不精确和 DECIMAL 精确计算代价高的问题。

4.1.3 字符串类型

MySQL 支持多种字符串类型，每种类型还有很多变种。这些数据类型在 4.1 和 5.0 版本

发生了很大的变化，使得情况更加复杂。从 MySQL 4.1 开始，每个字符串列可以定义自己的字符集和排序规则，或者说校对规则（collation）（更多关于这个主题的信息请参考第 7 章）。这些东西会很大程度上影响性能。

VARCHAR 和 CHAR 类型

VARCHAR 和 CHAR 是两种最主要的字符串类型。不幸的是，很难精确地解释这些值是怎么存储在磁盘和内存中的，因为这跟存储引擎的具体实现有关。下面的描述假设使用的存储引擎是 InnoDB 和 / 或者 MyISAM。如果使用的不是这两种存储引擎，请参考所使用的存储引擎的文档。

先看看 VARCHAR 和 CHAR 值通常在磁盘上怎么存储。请注意，存储引擎存储 CHAR 或者 VARCHAR 值的方式在内存中和在磁盘上可能不一样，所以 MySQL 服务器从存储引擎读出的值可能需要转换为另一种存储格式。下面是关于两种类型的一些比较。

VARCHAR

VARCHAR 类型用于存储可变长字符串，是最常见的字符串数据类型。它比定长类型更节省空间，因为它仅使用必要的空间（例如，越短的字符串使用越少的空间）。有一种情况例外，如果 MySQL 表使用 ROW_FORMAT=FIXED 创建的话，每一行都会使用定长存储，这会很浪费空间。

VARCHAR 需要使用 1 或 2 个额外字节记录字符串的长度：如果列的最大长度小于或等于 255 字节，则只使用 1 个字节表示，否则使用 2 个字节。假设采用 latin1 字符集，一个 VARCHAR(10) 的列需要 11 个字节的存储空间。VARCHAR(1000) 的列则需要 1002 个字节，因为需要 2 个字节存储长度信息。

VARCHAR 节省了存储空间，所以对性能也有帮助。但是，由于行是变长的，在 UPDATE 时可能使行变得比原来更长，这就导致需要做额外的工作。如果一个行占用的空间增长，并且在页内没有更多的空间可以存储，在这种情况下，不同的存储引擎的处理方式是不一样的。例如，MyISAM 会将行拆成不同的片段存储，InnoDB 则需要分裂页来使行可以放进页内。其他一些存储引擎也许从不在原数据位置更新数据。

下面这些情况下使用 VARCHAR 是合适的：字符串列的最大长度比平均长度大很多；列的更新很少，所以碎片不是问题；使用了像 UTF-8 这样复杂的字符集，每个字符都使用不同的字节数进行存储。 120

在 5.0 或者更高版本，MySQL 在存储和检索时会保留末尾空格。但在 4.1 或更老的版本，MySQL 会剔除末尾空格。

InnoDB 则更灵活，它可以把过长的 VARCHAR 存储为 BLOB，我们稍后讨论这个问题。

CHAR

CHAR 类型是定长的：MySQL 总是根据定义的字符串长度分配足够的空间。当存储 CHAR 值时，MySQL 会删除所有的末尾空格（在 MySQL 4.1 和更老版本中 VARCHAR 也是这样实现的——也就是说这些版本中 CHAR 和 VARCHAR 在逻辑上是一样的，区别只是在存储格式上）。CHAR 值会根据需要采用空格进行填充以方便比较。

CHAR 适合存储很短的字符串，或者所有值都接近同一个长度。例如，CHAR 非常适合存储密码的 MD5 值，因为这是一个定长的值。对于经常变更的数据，CHAR 也比 VARCHAR 更好，因为定长的 CHAR 类型不容易产生碎片。对于非常短的列，CHAR 比 VARCHAR 在存储空间上也更有效率。例如用 CHAR(1) 来存储只有 Y 和 N 的值，如果采用单字节字符集[注5] 只需要一个字节，但是 VARCHAR(1) 却需要两个字节，因为还有一个记录长度的额外字节。

CHAR 类型的这些行为可能有一点难以理解，下面通过一个具体的例子来说明。首先，我们创建一张只有一个 CHAR(10) 字段的表并且往里面插入一些值：

```
mysql> CREATE TABLE char_test( char_col CHAR(10));
mysql> INSERT INTO char_test(char_col) VALUES
    -> ('string1'), ('  string2'), ('string3 ');
```

当检索这些值的时候，会发现 string3 末尾的空格被截断了。

```
mysql> SELECT CONCAT("'", char_col, "'") FROM char_test;
+----------------------------+
| CONCAT("'", char_col, "'") |
+----------------------------+
| 'string1'                  |
| '  string2'                |
| 'string3'                  |
+----------------------------+
```

如果用 VARCHAR(10) 字段存储相同的值，可以得到如下结果[注6]：

121
```
mysql> SELECT CONCAT("'", varchar_col, "'") FROM varchar_test;
+-------------------------------+
| CONCAT("'", varchar_col, "'") |
+-------------------------------+
| 'string1'                     |
| '  string2'                   |
| 'string3 '                    |
+-------------------------------+
```

数据如何存储取决于存储引擎，并非所有的存储引擎都会按照相同的方式处理定长和变长的字符串。Memory 引擎只支持定长的行，即使有变长字段也会根据最大长度分

注 5： 记住字符串长度定义不是字节数，是字符数。多字节字符集会需要更多的空间存储单个字符。
注 6： string3 尾部的空格还在。——译者注

配最大空间[注7]。不过，填充和截取空格的行为在不同存储引擎都是一样的，因为这是在 MySQL 服务器层进行处理的。

与 CHAR 和 VARCHAR 类似的类型还有 BINARY 和 VARBINARY，它们存储的是二进制字符串。二进制字符串跟常规字符串非常相似，但是二进制字符串存储的是字节码而不是字符。填充也不一样：MySQL 填充 BINARY 采用的是 \0（零字节）而不是空格，在检索时也不会去掉填充值[注8]。

当需要存储二进制数据，并且希望 MySQL 使用字节码而不是字符进行比较时，这些类型是非常有用的。二进制比较的优势并不仅仅体现在大小写敏感上。MySQL 比较 BINARY 字符串时，每次按一个字节，并且根据该字节的数值进行比较。因此，二进制比较比字符比较简单很多，所以也就更快。

> **慷慨是不明智的**
>
> 使用 VARCHAR(5) 和 VARCHAR(200) 存储 'hello' 的空间开销是一样的。那么使用更短的列有什么优势吗？
>
> 事实证明有很大的优势。更长的列会消耗更多的内存，因为 MySQL 通常会分配固定大小的内存块来保存内部值。尤其是使用内存临时表进行排序或操作时会特别糟糕。在利用磁盘临时表进行排序时也同样糟糕。
>
> 所以最好的策略是只分配真正需要的空间。

BLOB 和 TEXT 类型

BLOB 和 TEXT 都是为存储很大的数据而设计的字符串数据类型，分别采用二进制和字符方式存储。

实际上，它们分别属于两组不同的数据类型家族：字符类型是 TINYTEXT，SMALLTEXT，TEXT，MEDIUMTEXT，LONGTEXT；对应的二进制类型是 TINYBLOB，SMALLBLOB，BLOB，MEDIUMBLOB，LONGBLOB。BLOB 是 SMALLBLOB 的同义词，TEXT 是 SMALLTEXT 的同义词。 122

与其他类型不同，MySQL 把每个 BLOB 和 TEXT 值当作一个独立的对象处理。存储引擎在存储时通常会做特殊处理。当 BLOB 和 TEXT 值太大时，InnoDB 会使用专门的“外部”

注 7： Percona Server 里的 Memory 引擎支持变长的行。

注 8： 如果需要在检索时保持值不变，则需要特别小心 BINARY 类型，MySQL 会用 \0 将其填充到需要的长度。

存储区域来进行存储，此时每个值在行内需要 1 ~ 4 个字节存储一个指针，然后在外部存储区域存储实际的值。

BLOB 和 TEXT 家族之间仅有的不同是 BLOB 类型存储的是二进制数据，没有排序规则或字符集，而 TEXT 类型有字符集和排序规则。

MySQL 对 BLOB 和 TEXT 列进行排序与其他类型是不同的：它只对每个列的最前 max_sort_length 字节而不是整个字符串做排序。如果只需要排序前面一小部分字符，则可以减小 max_sort_length 的配置，或者使用 ORDER BY SUSTRING(*column*, *length*)。

MySQL 不能将 BLOB 和 TEXT 列全部长度的字符串进行索引，也不能使用这些索引消除排序。(关于这个主题下一章会有更多的信息。)

磁盘临时表和文件排序

因为 Memory 引擎不支持 BLOB 和 TEXT 类型，所以，如果查询使用了 BLOB 或 TEXT 列并且需要使用隐式临时表，将不得不使用 MyISAM 磁盘临时表，即使只有几行数据也是如此（Percona Server 的 Memory 引擎支持 BLOB 和 TEXT 类型，但直到本书写作之际，同样的场景下还是需要使用磁盘临时表)。

这会导致严重的性能开销。即使配置 MySQL 将临时表存储在内存块设备上（RAM Disk)，依然需要许多昂贵的系统调用。

最好的解决方案是尽量避免使用 BLOB 和 TEXT 类型。如果实在无法避免，有一个技巧是在所有用到 BLOB 字段的地方都使用 SUBSTRING(*column*, *length*) 将列值转换为字符串（在 ORDER BY 子句中也适用)，这样就可以使用内存临时表了。但是要确保截取的子字符串足够短，不会使临时表的大小超过 max_heap_table_size 或 tmp_table_size，超过以后 MySQL 会将内存临时表转换为 MyISAM 磁盘临时表。

最坏情况下的长度分配对于排序的时候也是一样的，所以这一招对于内存中创建大临时表和文件排序，以及在磁盘上创建大临时表和文件排序这两种情况都很有帮助。

123 例如，假设有一个 1 000 万行的表，占用几个 GB 的磁盘空间。其中有一个 utf8 字符集的 VARCHAR(1000) 列。每个字符最多使用 3 个字节，最坏情况下需要 3 000 字节的空间。如果在 ORDER BY 中用到这个列，并且查询扫描整个表，为了排序就需要超过 30GB 的临时表。

如果 EXPLAIN 执行计划的 Extra 列包含"Using temporary"，则说明这个查询使用了隐式临时表。

使用枚举（ENUM）代替字符串类型

有时候可以使用枚举列代替常用的字符串类型。枚举列可以把一些不重复的字符串存储成一个预定义的集合。MySQL 在存储枚举时非常紧凑，会根据列表值的数量压缩到一个或者两个字节中。MySQL 在内部会将每个值在列表中的位置保存为整数，并且在表的 *.frm* 文件中保存"数字 - 字符串"映射关系的"查找表"。下面有一个例子：

```
mysql> CREATE TABLE enum_test(
    ->    e ENUM('fish', 'apple', 'dog') NOT NULL
    -> );
mysql> INSERT INTO enum_test(e) VALUES('fish'), ('dog'), ('apple');
```

这三行数据实际存储为整数，而不是字符串。可以通过在数字上下文环境检索看到这个双重属性：

```
mysql> SELECT e + 0 FROM enum_test;
+-------+
| e + 0 |
+-------+
|     1 |
|     3 |
|     2 |
+-------+
```

如果使用数字作为 ENUM 枚举常量，这种双重性很容易导致混乱，例如 ENUM('1','2','3')。建议尽量避免这么做。

另外一个让人吃惊的地方是，枚举字段是按照内部存储的整数而不是定义的字符串进行排序的：

```
mysql> SELECT e FROM enum_test ORDER BY e;
+-------+
| e     |
+-------+
| fish  |
| apple |
| dog   |
+-------+
```

一种绕过这种限制的方式是按照需要的顺序来定义枚举列。另外也可以在查询中使用 FIELD() 函数显式地指定排序顺序，但这会导致 MySQL 无法利用索引消除排序。 124

```
mysql> SELECT e FROM enum_test ORDER BY FIELD(e, 'apple', 'dog', 'fish');
+-------+
| e     |
+-------+
| apple |
| dog   |
| fish  |
+-------+
```

如果在定义时就是按照字母的顺序，就没有必要这么做了。

枚举最不好的地方是，字符串列表是固定的，添加或删除字符串必须使用 ALTER TABLE。因此，对于一系列未来可能会改变的字符串，使用枚举不是一个好主意，除非能接受只在列表末尾添加元素，这样在 MySQL 5.1 中就可以不用重建整个表来完成修改。

由于 MySQL 把每个枚举值保存为整数，并且必须进行查找才能转换为字符串，所以枚举列有一些开销。通常枚举的列表都比较小，所以开销还可以控制，但也不能保证一直如此。在特定情况下，把 CHAR/VARCHAR 列与枚举列进行关联可能会比直接关联 CHAR/VARCHAR 列更慢。

为了说明这个情况，我们对一个应用中的一张表进行了基准测试，看看在 MySQL 中执行上面说的关联的速度如何。该表有一个很大的主键：

```
CREATE TABLE webservicecalls (
   day date NOT NULL,
   account smallint NOT NULL,
   service varchar(10) NOT NULL,
   method varchar(50) NOT NULL,
   calls int NOT NULL,
   items int NOT NULL,
   time float NOT NULL,
   cost decimal(9,5) NOT NULL,
   updated datetime,
   PRIMARY KEY  (day, account, service, method)
) ENGINE=InnoDB;
```

这个表有 11 万行数据，只有 10MB 大小，所以可以完全载入内存。service 列包含了 5 个不同的值，平均长度为 4 个字符，method 列包含了 71 个值，平均长度为 20 个字符。

我们复制一下这个表，但是把 service 和 method 字段换成枚举类型，表结构如下：

125

```
CREATE TABLE webservicecalls_enum (
   ... omitted ...
   service ENUM(...values omitted...) NOT NULL,
   method ENUM(...values omitted...) NOT NULL,
   ... omitted ...
) ENGINE=InnoDB;
```

然后我们用主键列关联这两个表，下面是所使用的查询语句：

```
mysql> SELECT SQL_NO_CACHE COUNT(*)
    -> FROM webservicecalls
    ->    JOIN webservicecalls USING(day, account, service, method);
```

我们用 VARVHAR 和 ENUM 分别测试了这个语句，结果如表 4-1 所示。

表 4-1：连接VARCHAR和ENUM列的速度

测试	QPS
VARCHAR 关联 VARCHAR	2.6
VARCHAR 关联 ENUM	1.7
ENUM 关联 VARCHAR	1.8
ENUM 关联 ENUM	3.5

从上面的结果可以看到，当把列都转换成 ENUM 以后，关联变得很快。但是当 VARCHAR 列和 ENUM 列进行关联时则慢很多。在本例中，如果不是必须和 VARCHAR 列进行关联，那么转换这些列为 ENUM 就是个好主意。这是一个通用的设计实践，在“查找表”时采用整数主键而避免采用基于字符串的值进行关联。

然而，转换列为枚举型还有另一个好处。根据 SHOW TABLE STATUS 命令输出结果中 Data_length 列的值，把这两列转换为 ENUM 可以让表的大小缩小 1/3。在某些情况下，即使可能出现 ENUM 和 VARCHAR 进行关联的情况，这也是值得的[注9]。同样，转换后主键也只有原来的一半大小了。因为这是 InnoDB 表，如果表上有其他索引，减小主键大小会使非主键索引也变得更小。稍后再解释这个问题。

4.1.4 日期和时间类型

MySQL 可以使用许多类型来保存日期和时间值，例如 YEAR 和 DATE。MySQL 能存储的最小时间粒度为秒（MariaDB 支持微秒级别的时间类型）。但是 MySQL 也可以使用微秒级的粒度进行临时运算，我们会展示怎么绕开这种存储限制。

大部分时间类型都没有替代品，因此没有什么是最佳选择的问题。唯一的问题是保存日期和时间的时候需要做什么。MySQL 提供两种相似的日期类型：DATETIME 和 TIMESTAMP。对于很多应用程序，它们都能工作，但是在某些场景，一个比另一个工作得好。让我们来看一下。 126

DATETIME

这个类型能保存大范围的值，从 1001 年到 9999 年，精度为秒。它把日期和时间封

注 9： 这很可能可以节省 I/O。——译者注

装到格式为 YYYYMMDDHHMMSS 的整数中，与时区无关。使用 8 个字节的存储空间。

默认情况下，MySQL 以一种可排序的、无歧义的格式显示 DATETIME 值，例如“2008-01-16 22:37:08”。这是 ANSI 标准定义的日期和时间表示方法。

TIMESTAMP

就像它的名字一样，TIMETAMP 类型保存了从 1970 年 1 月 1 日午夜（格林尼治标准时间）以来的秒数，它和 UNIX 时间戳相同。TIMESTAMP 只使用 4 个字节的存储空间，因此它的范围比 DATETIME 小得多：只能表示从 1970 年到 2038 年。MySQL 提供了 FROM_UNIXTIME() 函数把 Unix 时间戳转换为日期，并提供了 UNIX_TIMESTAMP() 函数把日期转换为 Unix 时间戳。

MySQL 4.1 以及更新的版本按照 DATETIME 的方式格式化 TIMESTAMP 的值，但是 MySQL 4.0 以及更老的版本不会在各个部分之间显示任何标点符号。这仅仅是显示格式上的区别，TIMESTAMP 的存储格式在各个版本都是一样的。

TIMESTAMP 显示的值也依赖于时区。MySQL 服务器、操作系统，以及客户端连接都有时区设置。

因此，存储值为 0 的 TIMESTAMP 在美国东部时区显示为“1969-12-31 19:00:00”，与格林尼治时间差 5 个小时。有必要强调一下这个区别：如果在多个时区存储或访问数据，TIMESTAMP 和 DATETIME 的行为将很不一样。前者提供的值与时区有关系，后者则保留文本表示的日期和时间。

TIMESTAMP 也有 DATETIME 没有的特殊属性。默认情况下，如果插入时没有指定第一个 TIMESTAMP 列的值，MySQL 则设置这个列的值为当前时间[注10]。在插入一行记录时，MySQL 默认也会更新第一个 TIMESTAMP 列的值（除非在 UPDATE 语句中明确指定了值）。你可以配置任何 TIMESTAMP 列的插入和更新行为。最后，TIMESTAMP 列默认为 NOT NULL，这也和其他的数据类型不一样。

127 除了特殊行为之外，通常也应该尽量使用 TIMESTAMP，因为它比 DATETIME 空间效率更高。有时候人们会将 Unix 时间截存储为整数值，但这不会带来任何收益。用整数保存时间截的格式通常不方便处理，所以我们不推荐这样做。

如果需要存储比秒更小粒度的日期和时间值怎么办？MySQL 目前没有提供合适的数据类型，但是可以使用自己的存储格式：可以使用 BIGINT 类型存储微秒级别的时间截，或者使用 DOUBLE 存储秒之后的小数部分。这两种方式都可以，或者也可以使用 MariaDB 替代 MySQL。

注 10：TIMESTAMP 的行为规则比较复杂，并且在不同的 MySQL 版本里会变动，所以你应该验证数据库的行为是你需要的。一个好的方式是修改完 TIMESTAMP 列后用 SHOW CREATE TABLE 命令检查输出。

4.1.5 位数据类型

MySQL 有少数几种存储类型使用紧凑的位存储数据。所有这些位类型，不管底层存储格式和处理方式如何，从技术上来说都是字符串类型。

BIT

在 MySQL 5.0 之前，BIT 是 TINYINT 的同义词。但是在 MySQL 5.0 以及更新版本，这是一个特性完全不同的数据类型。下面我们将讨论 BIT 类型新的行为特性。

可以使用 BIT 列在一列中存储一个或多个 true/false 值。BIT(1) 定义一个包含单个位的字段，BIT(2) 存储 2 个位，依此类推。BIT 列的最大长度是 64 个位。

BIT 的行为因存储引擎而异。MyISAM 会打包存储所有的 BIT 列，所以 17 个单独的 BIT 列只需要 17 个位存储（假设没有可为 NULL 的列），这样 MyISAM 只使用 3 个字节就能存储这 17 个 BIT 列。其他存储引擎例如 Memory 和 InnoDB，为每个 BIT 列使用一个足够存储的最小整数类型来存放，所以不能节省存储空间。

MySQL 把 BIT 当作字符串类型，而不是数字类型。当检索 BIT(1) 的值时，结果是一个包含二进制 0 或 1 值的字符串，而不是 ASCII 码的“0”或“1”。然而，在数字上下文的场景中检索时，结果将是位字符串转换成的数字。如果需要和另外的值比较结果，一定要记得这一点。例如，如果存储一个值 b'00111001'（二进制值等于 57）到 BIT(8) 的列并且检索它，得到的内容是字符码为 57 的字符串。也就是说得到 ASCII 码为 57 的字符“9”。但是在数字上下文场景中，得到的是数字 57：

```
mysql> CREATE TABLE bittest(a bit(8));
mysql> INSERT INTO bittest VALUES(b'00111001');
mysql> SELECT a, a + 0 FROM bittest;
+------+-------+
| a    | a + 0 |
+------+-------+
| 9    |    57 |
+------+-------+
```

128

这是相当令人费解的，所以我们认为应该谨慎使用 BIT 类型。对于大部分应用，最好避免使用这种类型。

如果想在一个 bit 的存储空间中存储一个 true/false 值，另一个方法是创建一个可以为空的 CHAR(0) 列。该列可以保存空值（NULL）或者长度为零的字符串（空字符串）。

SET

如果需要保存很多 true/false 值，可以考虑合并这些列到一个 SET 数据类型，它在 MySQL 内部是以一系列打包的位的集合来表示的。这样就有效地利用了存储空间，并且 MySQL 有像 FIND_IN_SET() 和 FIELD() 这样的函数，方便地在查询中使用。它的主要缺点是改变列的定义的代价较高：需要 ALTER TABLE，这对大表来说是非

常昂贵的操作（但是本章的后面给出了解决办法）。一般来说，也无法在 SET 列上通过索引查找。

在整数列上进行按位操作

一种替代 SET 的方式是使用一个整数包装一系列的位。例如，可以把 8 个位包装到一个 TINYINT 中，并且按位操作来使用。可以在应用中为每个位定义名称常量来简化这个工作。

比起 SET,这种办法主要的好处在于可以不使用 ALTER TABLE 改变字段代表的“枚举”值,缺点是查询语句更难写,并且更难理解（当第 5 个 bit 位被设置时是什么意思？）。一些人非常适应这种方式，也有一些人不适应，所以是否采用这种技术取决于个人的偏好。

一个包装位的应用的例子是保存权限的访问控制列表（ACL）。每个位或者 SET 元素代表一个值，例如 CAN_READ、CAN_WRITE，或者 CAN_DELETE。如果使用 SET 列，可以让 MySQL 在列定义里存储位到值的映射关系；如果使用整数列，则可以在应用代码里存储这个对应关系。这是使用 SET 列时的查询：

```
mysql> CREATE TABLE acl (
    ->    perms SET('CAN_READ', 'CAN_WRITE', 'CAN_DELETE') NOT NULL
    -> );
mysql> INSERT INTO acl(perms) VALUES ('CAN_READ,CAN_DELETE');
mysql> SELECT perms FROM acl WHERE FIND_IN_SET('AN_READ', perms);
+---------------------+
| perms               |
+---------------------+
| CAN_READ,CAN_DELETE |
+---------------------+
```

如果使用整数来存储，则可以参考下面的例子：

129

```
mysql> SET @CAN_READ   := 1 << 0,
    ->     @CAN_WRITE  := 1 << 1,
    ->     @CAN_DELETE := 1 << 2;
mysql> CREATE TABLE acl (
    ->    perms TINYINT UNSIGNED NOT NULL DEFAULT 0
    -> );
mysql> INSERT INTO acl(perms) VALUES(@CAN_READ + @CAN_DELETE);
mysql> SELECT perms FROM acl WHERE perms & @CAN_READ;
+-------+
| perms |
+-------+
|     5 |
+-------+
```

这里我们使用 MySQL 变量来定义值，但是也可以在代码里使用常量来代替。

4.1.6 选择标识符（identifier）

为标识列（identifier column）选择合适的数据类型非常重要。一般来说更有可能用标识列与其他值进行比较（例如，在关联操作中），或者通过标识列寻找其他列。标识列也可能在另外的表中作为外键使用，所以为标识列选择数据类型时，应该选择跟关联表中的对应列一样的类型（正如我们在本章早些时候所论述的一样，在相关的表中使用相同的数据类型是个好主意，因为这些列很可能在关联中使用）。

当选择标识列的类型时，不仅仅需要考虑存储类型，还需要考虑 MySQL 对这种类型怎么执行计算和比较。例如，MySQL 在内部使用整数存储 ENUM 和 SET 类型，然后在做比较操作时转换为字符串。

一旦选定了一种类型，要确保在所有关联表中都使用同样的类型。类型之间需要精确匹配，包括像 UNSIGNED 这样的属性[注11]。混用不同数据类型可能导致性能问题，即使没有性能影响，在比较操作时隐式类型转换也可能导致很难发现的错误。这种错误可能会很久以后才突然出现，那时候可能都已经忘记是在比较不同的数据类型。

在可以满足值的范围的需求，并且预留未来增长空间的前提下，应该选择最小的数据类型。例如有一个 state_id 列存储美国各州的名字[注12]，就不需要几千或几百万个值，所以不需要使用 INT。TINYINT 足够存储，而且比 INT 少了 3 个字节。如果用这个值作为其他表的外键，3 个字节可能导致很大的性能差异。下面是一些小技巧。

整数类型 130

整数通常是标识列最好的选择，因为它们很快并且可以使用 AUTO_INCREMENT。

ENUM 和 SET 类型

对于标识列来说，EMUM 和 SET 类型通常是一个糟糕的选择，尽管对某些只包含固定状态或者类型的静态"定义表"来说可能是没有问题的。ENUM 和 SET 列适合存储固定信息，例如有序的状态、产品类型、人的性别。

举个例子，如果使用枚举字段来定义产品类型，也许会设计一张以这个枚举字段为主键的查找表（可以在查找表中增加一些列来保存描述性质的文本，这样就能够生成一个术语表，或者为网站的下拉菜单提供有意义的标签）。这时，使用枚举类型作为标识列是可行的，但是大部分情况下都要避免这么做。

字符串类型

如果可能，应该避免使用字符串类型作为标识列，因为它们很消耗空间，并且通

注 11：如果使用的是 InnoDB 存储引擎，将不能在数据类型不是完全匹配的情况下创建外键，否则会有报错信息："ERROR 1005 (HY000): Can't create table"，这个信息可能让人迷惑不解，这个问题在 MySQL 邮件组也经常有人抱怨（但奇怪的是，在不同长度的 VARCHAR 列上创建外键又是可以的）。

注 12：这是关联到另一张存储名字的表的 ID。——译者注

常比数字类型慢。尤其是在 MyISAM 表里使用字符串作为标识列时要特别小心。MyISAM 默认对字符串使用压缩索引，这会导致查询慢得多。在我们的测试中，我们注意到最多有 6 倍的性能下降。

对于完全“随机”的字符串也需要多加注意，例如 MD5()、SHA1() 或者 UUID() 产生的字符串。这些函数生成的新值会任意分布在很大的空间内，这会导致 INSERT 以及一些 SELECT 语句变得很慢[注13]：

- 因为插入值会随机地写到索引的不同位置，所以使得 INSERT 语句更慢。这会导致页分裂、磁盘随机访问，以及对于聚簇存储引擎产生聚簇索引碎片。关于这一点第 5 章有更多的讨论。
- SELECT 语句会变得更慢，因为逻辑上相邻的行会分布在磁盘和内存的不同地方。
- 随机值导致缓存对所有类型的查询语句效果都很差，因为会使得缓存赖以工作的访问局部性原理失效。如果整个数据集都一样的“热”，那么缓存任何一部分特定数据到内存都没有好处；如果工作集比内存大，缓存将会有很多刷新和不命中。

如果存储 UUID 值，则应该移除“-”符号；或者更好的做法是，用 UNHEX() 函数转换 UUID 值为 16 字节的数字，并且存储在一个 BINARY(16) 列中。检索时可以通过 HEX() 函数来格式化为十六进制格式。

131 UUID() 生成的值与加密散列函数例如 SHA1() 生成的值有不同的特征：UUID 值虽然分布也不均匀，但还是有一定顺序的。尽管如此，但还是不如递增的整数好用。

当心自动生成的 schema

我们已经介绍了大部分重要数据类型的考虑（有些会严重影响性能，有些则影响较小），但是我们还没有提到自动生成的 schema 设计有多么糟糕。

写得很烂的 schema 迁移程序，或者自动生成 schema 的程序，都会导致严重的性能问题。有些程序存储任何东西都会使用很大的 VARCHAR 列，或者对需要在关联时比较的列使用不同的数据类型。如果 schema 是自动生成的，一定要反复检查确认没有问题。

对象关系映射（ORM）系统（以及使用它们的“框架”）是另一种常见的性能噩梦。一些 ORM 系统会存储任意类型的数据到任意类型的后端数据存储中，这通常意味

注 13： 另一方面，对一些有很多写的特别大的表，这种伪随机值实际上可以帮助消除热点。

着其没有设计使用更优的数据类型来存储。有时会为每个对象的每个属性使用单独的行，甚至使用基于时间戳的版本控制，导致单个属性会有多个版本存在。

这种设计对开发者很有吸引力，因为这使得他们可以用面向对象的方式工作，不需要考虑数据是怎么存储的。然而，"对开发者隐藏复杂性"的应用通常不能很好地扩展。我们建议在用性能交换开发人员的效率之前仔细考虑，并且总是在真实大小的数据集上做测试，这样就不会太晚才发现性能问题。

4.1.7 特殊类型数据

某些类型的数据并不直接与内置类型一致。低于秒级精度的时间戳就是一个例子；本章的前面部分也演示过存储此类数据的一些选项。

另一个例子是一个 IPv4 地址。人们经常使用 `VARCHAR(15)` 列来存储 IP 地址。然而，它们实际上是 32 位无符号整数，不是字符串。用小数点将地址分成四段的表示方法只是为了让人们阅读容易。所以应该用无符号整数存储 IP 地址。MySQL 提供 `INET_ATON()` 和 `INET_NTOA()` 函数在这两种表示方法之间转换。

4.2 MySQL schema 设计中的陷阱

虽然有一些普遍的好或坏的设计原则，但也有一些问题是由 MySQL 的实现机制导致的，这意味着有可能犯一些只在 MySQL 下发生的特定错误。本节我们讨论设计 MySQL 的 schema 的问题。这也许会帮助你避免这些错误，并且选择在 MySQL 特定实现下工作得更好的替代方案。 132

太多的列

MySQL 的存储引擎 API 工作时需要在服务器层和存储引擎层之间通过行缓冲格式拷贝数据，然后在服务器层将缓冲内容解码成各个列。从行缓冲中将编码过的列转换成行数据结构的操作代价是非常高的。MyISAM 的定长行结构实际上与服务器层的行结构正好匹配，所以不需要转换。然而，MyISAM 的变长行结构和 InnoDB 的行结构则总是需要转换。转换的代价依赖于列的数量。当我们研究一个 CPU 占用非常高的案例时，发现客户使用了非常宽的表（数千个字段），然而只有一小部分列会实际用到，这时转换的代价就非常高。如果计划使用数千个字段，必须意识到服务器的性能运行特征会有一些不同。

太多的关联

所谓的“实体 - 属性 - 值”（EAV）设计模式是一个常见的糟糕设计模式，尤其是在 MySQL 下不能靠谱地工作。MySQL 限制了每个关联操作最多只能有 61 张表，但是 EAV 数据库需要许多自关联。我们见过不少 EAV 数据库最后超过了这个限制。事实上在许多关联少于 61 张表的情况下，解析和优化查询的代价也会成为 MySQL 的问题。一个粗略的经验法则，如果希望查询执行得快速且并发性好，单个查询最好在 12 个表以内做关联。

全能的枚举

注意防止过度使用枚举（ENUM）。下面是我们见过的一个例子：

```
CREATE TABLE ... (
  country enum('','0','1','2',...,'31')
```

这种模式的 schema 设计非常凌乱。这么使用枚举值类型也许在任何支持枚举类型的数据库都是一个有问题的设计方案，这里应该用整数作为外键关联到字典表或者查找表来查找具体值。但是在 MySQL 中，当需要在枚举列表中增加一个新的国家时就要做一次 ALTER TABLE 操作。在 MySQL 5.0 以及更早的版本中 ALTER TABLE 是一种阻塞操作；即使在 5.1 和更新版本中，如果不是在列表的末尾增加值也会一样需要 ALTER TABLE（我们将展示一些骇客式的方法来避免阻塞操作，但是这只是骇客的玩法，别轻易用在生产环境中）。

变相的枚举

枚举（ENUM）列允许在列中存储一组定义值中的单个值，集合（SET）列则允许在列中存储一组定义值中的一个或多个值。有时候这可能比较容易导致混乱。这是一个例子：

```
CREATE TABLE ...(
  is_default set('Y','N') NOT NULL default 'N'
```

133 如果这里真和假两种情况不会同时出现，那么毫无疑问应该使用枚举列代替集合列。

非此发明（Not Invent Here）的 NULL

我们之前写了避免使用 NULL 的好处，并且建议尽可能地考虑替代方案。即使需要存储一个事实上的“空值”到表中时，也不一定非得使用 NULL。也许可以使用 0、某个特殊值，或者空字符串作为代替。

但是遵循这个原则也不要走极端。当确实需要表示未知值时也不要害怕使用 NULL。在一些场景中，使用 NULL 可能会比某个神奇常数更好。从特定类型的值域中选择一个不可能的值，例如用 -1 代表一个未知的整数，可能导致代码复杂很多，并容易引入 bug，还可能会让事情变得一团糟。处理 NULL 确实不容易，但有时候会比它的替代方案更好。

下面是一个我们经常看到的例子：

```
CREATE TABLE ... (
  dt DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00'
```

伪造的全0值可能导致很多问题（可以配置MySQL的SQL_MODE来禁止不可能的日期，对于新应用这是个非常好的实践经验，它不会让创建的数据库里充满不可能的值）。值得一提的是，MySQL 会在索引中存储 NULL 值，而 Oracle 则不会。

4.3 范式和反范式

对于任何给定的数据通常都有很多种表示方法，从完全的范式化到完全的反范式化，以及两者的折中。在范式化的数据库中，每个事实数据会出现并且只出现一次。相反，在反范式化的数据库中，信息是冗余的，可能会存储在多个地方。

如果不熟悉范式，则应该先学习一下。有很多这方面的不错的书和在线资源；在这里，我们只是给出阅读本章所需要的这方面的简单介绍。下面以经典的“雇员，部门，部门领导”的例子开始：

EMPLOYEE	DEPARTMENT	HEAD
Jones	Accounting	Jones
Smith	Engineering	Smith
Brown	Accounting	Jones
Green	Engineering	Smith

这个 schema 的问题是修改数据时可能发生不一致。假如 Say Brown 接任 Accounting 部 134
门的领导，需要修改多行数据来反映这个变化，这是很痛苦的事并且容易引入错误。如果“Jones”这一行显示部门的领导跟“Brown”这一行的不一样，就没有办法知道哪个是对的。这就像是有句老话说的：“一个人有两块手表就永远不知道时间”。此外，这个设计在没有雇员信息的情况下就无法表示一个部门——如果我们删除了所有 Accounting 部门的雇员，我们就失去了关于这个部门本身的所有记录。要避免这个问题，我们需要对这个表进行范式化，方式是拆分雇员和部门项。拆分以后可以用下面两张表分别来存储雇员表：

EMPLOYEE_NAME	DEPARTMENT
Jones	Accounting
Smith	Engineering
Brown	Accounting
Green	Engineering

和部门表：

DEPARTMENT	HEAD
Accounting	Jones
Engineering	Smith

这样设计的两张表符合第二范式，在很多情况下做到这一步已经足够好了。然而，第二范式只是许多可能的范式中的一种。

> 这个例子中我们使用姓（Last Name）作为主键，因为这是数据的“自然标识”。从实践来看，无论如何都不应该这么用。这既不能保证唯一性，而且用一个很长的字符串作为主键是很糟糕的主意。

4.3.1 范式的优点和缺点

当为性能问题而寻求帮助时，经常会被建议对 schema 进行范式化设计，尤其是写密集的场景。这通常是个好建议。因为下面这些原因，范式化通常能够带来好处：

- 范式化的更新操作通常比反范式化要快。
- 当数据较好地范式化时，就只有很少或者没有重复数据，所以只需要修改更少的数据。
- 范式化的表通常更小，可以更好地放在内存里，所以执行操作会更快。
- 135 很少有多余的数据意味着检索列表数据时更少需要 DISTINCT 或者 GROUP BY 语句。还是前面的例子：在非范式化的结构中必须使用 DISTINCT 或者 GROUP BY 才能获得一份唯一的部门列表，但是如果部门（DEPARTMENT）是一张单独的表，则只需要简单的查询这张表就行了。

范式化设计的 schema 的缺点是通常需要关联。稍微复杂一些的查询语句在符合范式的 schema 上都可能需要至少一次关联，也许更多。这不但代价昂贵，也可能使一些索引策略无效。例如，范式化可能将列存放在不同的表中，而这些列如果在一个表中本可以属于同一个索引。

4.3.2 反范式的优点和缺点

反范式化的 schema 因为所有数据都在一张表中，可以很好地避免关联。

如果不需要关联表，则对大部分查询最差的情况——即使表没有使用索引——是全表扫描。当数据比内存大时这可能比关联要快得多，因为这样避免了随机 I/O[注14]。

注 14： 全表扫描基本上是顺序 I/O，但也不是 100% 的，跟引擎的实现有关。——译者注

单独的表也能使用更有效的索引策略。假设有一个网站，允许用户发送消息，并且一些用户是付费用户。现在想查看付费用户最近的 10 条信息。如果是范式化的结构并且索引了发送日期字段 published，这个查询也许看起来像这样：

```
mysql> SELECT message_text, user_name
    -> FROM message
    ->    INNER JOIN user ON message.user_id=user.id
    -> WHERE user.account_type='premiumv'
    -> ORDER BY message.published DESC LIMIT 10;
```

要更有效地执行这个查询，MySQL 需要扫描 message 表的 published 字段的索引。对于每一行找到的数据，将需要到 user 表里检查这个用户是不是付费用户。如果只有一小部分用户是付费账户，那么这是效率低下的做法。

另一种可能的执行计划是从 user 表开始，选择所有的付费用户，获得他们所有的信息，并且排序。但这可能更加糟糕。

主要问题是关联，使得需要在一个索引中又排序又过滤。如果采用反范式化组织数据，将两张表的字段合并一下，并且增加一个索引（account_type, published)，就可以不通过关联写出这个查询。这将非常高效：

```
mysql> SELECT message_text,user_name
    -> FROM user_messages
    -> WHERE account_type='premium'
    -> ORDER BY published DESC
    -> LIMIT 10;
```

4.3.3 混用范式化和反范式化 136

范式化和反范式化的 schema 各有优劣，怎么选择最佳的设计？

事实是，完全的范式化和完全的反范式化 schema 都是实验室里才有的东西：在真实世界中很少会这么极端地使用。在实际应用中经常需要混用，可能使用部分范式化的 schema、缓存表，以及其他技巧。

最常见的反范式化数据的方法是复制或者缓存，在不同的表中存储相同的特定列。在 MySQL 5.0 和更新版本中，可以使用触发器更新缓存值，这使得实现这样的方案变得更简单。

在我们的网站实例中，可以在 user 表和 message 表中都存储 account_type 字段，而不用完全的反范式化。这避免了完全反范式化的插入和删除问题，因为即使没有消息的时候也绝不会丢失用户的信息。这样也不会把 user_message 表搞得太大，有利于高效地获取数据。

但是现在更新用户的账户类型的操作代价就高了，因为需要同时更新两张表。至于这会不会是一个问题，需要考虑更新的频率以及更新的时长，并和执行 SELECT 查询的频率进行比较。

另一个从父表冗余一些数据到子表的理由是排序的需要。例如，在范式化的 schema 里通过作者的名字对消息做排序的代价将会非常高，但是如果在 message 表中缓存 author_name 字段并且建好索引，则可以非常高效地完成排序。

缓存衍生值也是有用的。如果需要显示每个用户发了多少消息（像很多论坛做的），可以每次执行一个昂贵的子查询来计算并显示它；也可以在 user 表中建一个 num_messages 列，每当用户发新消息时更新这个值。

4.4 缓存表和汇总表

有时提升性能最好的方法是在同一张表中保存衍生的冗余数据。然而，有时也需要创建一张完全独立的汇总表或缓存表（特别是为满足检索的需求时）。如果能容许少量的脏数据，这是非常好的方法，但是有时确实没有选择的余地（例如，需要避免复杂、昂贵的实时更新操作）。

术语“缓存表”和“汇总表”没有标准的含义。我们用术语“缓存表”来表示存储那些可以比较简单地从 schema 其他表获取（但是每次获取的速度比较慢）数据的表（例如，
137 逻辑上冗余的数据）。而术语“汇总表”时，则保存的是使用 GROUP BY 语句聚合数据的表（例如，数据不是逻辑上冗余的）。也有人使用术语“累积表（Roll-Up Table）”称呼这些表。因为这些数据被“累积”了。

仍然以网站为例，假设需要计算之前 24 小时内发送的消息数。在一个很繁忙的网站不可能维护一个实时精确的计数器。作为替代方案，可以每小时生成一张汇总表。这样也许一条简单的查询就可以做到，并且比实时维护计数器要高效得多。缺点是计数器并不是 100% 精确。

如果必须获得过去 24 小时准确的消息发送数量（没有遗漏），有另外一种选择。以每小时汇总表为基础，把前 23 个完整的小时的统计表中的计数全部加起来，最后再加上开始阶段和结束阶段不完整的小时内的计数。假设统计表叫作 msg_per_hr 并且这样定义：

```
CREATE TABLE msg_per_hr (
   hr DATETIME NOT NULL,
   cnt INT UNSIGNED NOT NULL,
   PRIMARY KEY(hr)
);
```

可以通过把下面的三个语句的结果加起来，得到过去 24 小时发送消息的总数。我们使用 LEFT(NOW(),14) 来获得当前的日期和时间最接近的小时：

```
mysql> SELECT SUM(cnt) FROM msg_per_hr
    -> WHERE hr BETWEEN
    ->    CONCAT(LEFT(NOW(), 14), '00:00') - INTERVAL 23 HOUR
    ->    AND CONCAT(LEFT(NOW(), 14), '00:00') - INTERVAL 1 HOUR;
mysql> SELECT COUNT(*) FROM message
    -> WHERE posted >= NOW() - INTERVAL 24 HOUR
    ->    AND posted < CONCAT(LEFT(NOW(), 14), '00:00') - INTERVAL 23 HOUR;
mysql> SELECT COUNT(*) FROM message
    -> WHERE posted >= CONCAT(LEFT(NOW(), 14), '00:00');
```

不管是哪种方法——不严格的计数或通过小范围查询填满间隙的严格计数——都比计算 message 表的所有行要有效得多。这是建立汇总表的最关键原因。实时计算统计值是很昂贵的操作，因为要么需要扫描表中的大部分数据，要么查询语句只能在某些特定的索引上才能有效运行，而这类特定索引一般会对 UPDATE 操作有影响，所以一般不希望创建这样的索引。计算最活跃的用户或者最常见的“标签”是这种操作的典型例子。

缓存表则相反，其对优化搜索和检索查询语句很有效。这些查询语句经常需要特殊的表和索引结构，跟普通 OLTP 操作用的表有些区别。

例如，可能会需要很多不同的索引组合来加速各种类型的查询。这些矛盾的需求有时需要创建一张只包含主表中部分列的缓存表。一个有用的技巧是对缓存表使用不同的存储引擎。例如，如果主表使用 InnoDB，用 MyISAM 作为缓存表的引擎将会得到更小的索引占用空间，并且可以做全文搜索。有时甚至想把整个表导出 MySQL，插入到专门的搜索系统中获得更高的搜索效率，例如 Lucene 或者 Sphinx 搜索引擎。 138

在使用缓存表和汇总表时，必须决定是实时维护数据还是定期重建。哪个更好依赖于应用程序，但是定期重建并不只是节省资源，也可以保持表不会有很多碎片，以及有完全顺序组织的索引（这会更加高效）。

当重建汇总表和缓存表时，通常需要保证数据在操作时依然可用。这就需要通过使用“影子表”来实现，“影子表”指的是一张在真实表“背后”创建的表。当完成了建表操作后，可以通过一个原子的重命名操作切换影子表和原表。例如，如果需要重建 my_summary，则可以先创建 my_summary_new，然后填充好数据，最后和真实表做切换：

```
mysql> DROP TABLE IF EXISTS my_summary_new, my_summary_old;
mysql> CREATE TABLE my_summary_new LIKE my_summary;
-- populate my_summary_new as desired
mysql> RENAME TABLE my_summary TO my_summary_old, my_summary_new TO my_summary;
```

如果像上面的例子一样，在将 my_summary 这个名字分配给新建的表之前将原始的 my_summary 表重命名为 my_summary_old，就可以在下一次重建之前一直保留旧版本的数据。如果新表有问题，则可以很容易地进行快速回滚操作。

4.4.1 物化视图

许多数据库管理系统（例如 Oracle 或者微软 SQL Server）都提供了一个被称作物化视图的功能。物化视图实际上是预先计算并且存储在磁盘上的表，可以通过各种各样的策略刷新和更新。MySQL 并不原生支持物化视图（我们将在第 7 章详细探讨支持这种视图的细节）。然而，使用 Justin Swanhart 的开源工具 Flexviews（*http://code.google.com/p/flexviews/*），也可以自己实现物化视图。Flexviews 比完全自己实现的解决方案要更精细，并且提供了很多不错的功能使得可以更简单地创建和维护物化视图。它由下面这些部分组成：

- 变更数据抓取（Change Data Capture，CDC）功能，可以读取服务器的二进制日志并且解析相关行的变更。
- 一系列可以帮助创建和管理视图的定义的存储过程。
- 一些可以应用变更到数据库中的物化视图的工具。

139 对比传统的维护汇总表和缓存表的方法，Flexviews 通过提取对源表的更改，可以增量地重新计算物化视图的内容。这意味着不需要通过查询原始数据来更新视图。例如，如果创建了一张汇总表用于计算每个分组的行数，此后增加了一行数据到源表中，Flexviews 简单地给相应的组的行数加一即可。同样的技术对其他的聚合函数也有效，例如 SUM() 和 AVG()。这实际上是有好处的，基于行的二进制日志包含行更新前后的镜像，所以 Flexviews 不仅仅可以获得每行的新值，还可以不需要查找源表就能知道每行数据的旧版本。计算增量数据比从源表中读取数据的效率要高得多。

因为版面的限制，这里我们不会完整地探讨怎么使用 Flexviews，但是可以给出一个概略。先写出一个 SELECT 语句描述想从已经存在的数据库中得到的数据。这可能包含关联和聚合（GROUP BY）。Flexviews 中有一个辅助工具可以转换 SQL 语句到 Flexviews 的 API 调用。Flexviews 会做完所有的脏活、累活：监控数据库的变更并且转换后用于更新存储物化视图的表。现在应用可以简单地查询物化视图来替代查询需要检索的表。

Flexviews 有不错的 SQL 覆盖范围，包括一些棘手的表达式，你可能没有料到一个工具可以在 MySQL 服务器之外处理这些工作。这一点对创建基于复杂 SQL 表达式的视图很有用，可以用基于物化视图的简单、快速的查询替换原来复杂的查询。

4.4.2 计数器表

如果应用在表中保存计数器，则在更新计数器时可能碰到并发问题。计数器表在 Web 应用中很常见。可以用这种表缓存一个用户的朋友数、文件下载次数等。创建一张独立的表存储计数器通常是个好主意，这样可使计数器表小且快。使用独立的表可以帮助避免查询缓存失效，并且可以使用本节展示的一些更高级的技巧。

应该让事情变得尽可能简单，假设有一个计数器表，只有一行数据，记录网站的点击次数：

```
mysql> CREATE TABLE hit_counter (
    ->    cnt int unsigned not null
    -> ) ENGINE=InnoDB;
```

网站的每次点击都会导致对计数器进行更新：

```
mysql> UPDATE hit_counter SET cnt = cnt + 1;
```

问题在于，对于任何想要更新这一行的事务来说，这条记录上都有一个全局的互斥锁 140
(mutex)。这会使得这些事务只能串行执行。要获得更高的并发更新性能，也可以将计数器保存在多行中，每次随机选择一行进行更新。这样做需要对计数器表进行如下修改：

```
mysql> CREATE TABLE hit_counter (
    ->    slot tinyint unsigned not null primary key,
    ->    cnt int unsigned not null
    -> ) ENGINE=InnoDB;
```

然后预先在这张表增加 100 行数据。现在选择一个随机的槽（slot）进行更新：

```
mysql> UPDATE hit_counter SET cnt = cnt + 1 WHERE slot = RAND() * 100;
```

要获得统计结果，需要使用下面这样的聚合查询：

```
mysql> SELECT SUM(cnt) FROM hit_counter;
```

一个常见的需求是每隔一段时间开始一个新的计数器（例如，每天一个）。如果需要这么做，则可以再简单地修改一下表设计：

```
mysql> CREATE TABLE daily_hit_counter (
    ->    day date not null,
    ->    slot tinyint unsigned not null,
    ->    cnt int unsigned not null,
    ->    primary key(day, slot)
    -> ) ENGINE=InnoDB;
```

在这个场景中，可以不用像前面的例子那样预先生成行，而用 ON DUPLICATE KEY UPDATE 代替：

```
mysql> INSERT INTO daily_hit_counter(day, slot, cnt)
    ->    VALUES(CURRENT_DATE, RAND() * 100, 1)
    ->    ON DUPLICATE KEY UPDATE cnt = cnt + 1;
```

如果希望减少表的行数，以避免表变得太大，可以写一个周期执行的任务，合并所有结果到 0 号槽，并且删除所有其他的槽：

```
mysql> UPDATE daily_hit_counter as c
    ->    INNER JOIN (
    ->       SELECT day, SUM(cnt) AS cnt, MIN(slot) AS mslot
    ->       FROM daily_hit_counter
    ->       GROUP BY day
    ->    ) AS x USING(day)
    -> SET c.cnt  = IF(c.slot = x.mslot, x.cnt, 0),
    ->     c.slot = IF(c.slot = x.mslot, 0, c.slot);
mysql> DELETE FROM daily_hit_counter WHERE slot <> 0 AND cnt = 0;
```

141 **更快地读，更慢地写**

为了提升读查询的速度，经常会需要建一些额外索引，增加冗余列，甚至是创建缓存表和汇总表。这些方法会增加写查询的负担，也需要额外的维护任务，但在设计高性能数据库时，这些都是常见的技巧：虽然写操作变得更慢了，但更显著地提高了读操作的性能。

然而，写操作变慢并不是读操作变得更快所付出的唯一代价，还可能同时增加了读操作和写操作的开发难度。

4.5 加快 ALTER TABLE 操作的速度

MySQL 的 ALTER TABLE 操作的性能对大表来说是个大问题。MySQL 执行大部分修改表结构操作的方法是用新的结构创建一个空表，从旧表中查出所有数据插入新表，然后删除旧表。这样操作可能需要花费很长时间，如果内存不足而表又很大，而且还有很多索引的情况下尤其如此。许多人都有这样的经验，ALTER TABLE 操作需要花费数个小时甚至数天才能完成。

MySQL 5.1 以及更新版本包含一些类型的“在线”操作的支持，这些功能不需要在整个操作过程中锁表。最近版本的 InnoDB[注15] 也支持通过排序来建索引，这使得建索引更快并且有一个紧凑的索引布局。

注 15：就是所谓的“InnoDB plugin”，MySQL 5.5 和更新版本中唯一的 InnoDB。请参考第 1 章中关于 InnoDB 发布历史的细节。

一般而言，大部分 ALTER TABLE 操作将导致 MySQL 服务中断。我们会展示一些在 DDL 操作时有用的技巧，但这是针对一些特殊的场景而言的。对常见的场景，能使用的技巧只有两种：一种是先在一台不提供服务的机器上执行 ALTER TABLE 操作，然后和提供服务的主库进行切换；另外一种技巧是"影子拷贝"。影子拷贝的技巧是用要求的表结构创建一张和源表无关的新表，然后通过重命名和删表操作交换两张表。也有一些工具可以帮助完成影子拷贝工作：例如，Facebook 数据库运维团队（*https://launchpad.net/mysqlatfacebook*）的"online schema change"工具、Shlomi Noach 的 openark toolkit（*http://code.openark.org/*），以及 Percona Toolkit（*http://www.percona.com/software/*）。如果使用 Flexviews（参考 4.4.1 节），也可以通过其 CDC 工具执行无锁的表结构变更。

不是所有的 ALTER TABLE 操作都会引起表重建。例如，有两种方法可以改变或者删除一个列的默认值（一种方法很快，另外一种则很慢）。假如要修改电影的默认租赁期限，从三天改到五天。下面是很慢的方式：

```
mysql> ALTER TABLE sakila.film
    -> MODIFY COLUMN rental_duration TINYINT(3) NOT NULL DEFAULT 5;
```

SHOW STATUS 显示这个语句做了 1 000 次读和 1 000 次插入操作。换句话说，它拷贝了整张表到一张新表，甚至列的类型、大小和可否为 NULL 属性都没改变。 142

理论上，MySQL 可以跳过创建新表的步骤。列的默认值实际上存在表的 *.frm* 文件中，所以可以直接修改这个文件而不需要改动表本身。然而 MySQL 还没有采用这种优化的方法，所有的 MODIFY COLUMN 操作都将导致表重建。

另外一种方法是通过 ALTER COLUMN[注16] 操作来改变列的默认值：

```
mysql> ALTER TABLE sakila.film
    -> ALTER COLUMN rental_duration SET DEFAULT 5;
```

这个语句会直接修改 *.frm* 文件而不涉及表数据。所以，这个操作是非常快的。

4.5.1 只修改 .frm 文件

从上面的例子我们看到修改表的 *.frm* 文件是很快的，但 MySQL 有时候会在没有必要的时候也重建表。如果愿意冒一些风险，可以让 MySQL 做一些其他类型的修改而不用重建表。

注 16： ALTER TABLE 允许使用 ALTER COLUMN、MODIFY COLUMN 和 CHANGE COLUMN 语句修改列。这三种操作都是不一样的。

> 我们下面要演示的技巧是不受官方支持的，也没有文档记录，并且也可能不能正常工作，采用这些技术需要自己承担风险。建议在执行之前首先备份数据！

下面这些操作是有可能不需要重建表的：

- 移除（不是增加）一个列的 AUTO_INCREMENT 属性。
- 增加、移除，或更改 ENUM 和 SET 常量。如果移除的是已经有行数据用到其值的常量，查询将会返回一个空字串值。

基本的技术是为想要的表结构创建一个新的 *.frm* 文件，然后用它替换掉已经存在的那张表的 *.frm* 文件，像下面这样：

1. 创建一张有相同结构的空表，并进行所需要的修改（例如增加 ENUM 常量）。
2. 执行 FLUSH TABLES WITH READ LOCK。这将会关闭所有正在使用的表，并且禁止任何表被打开。
3. 交换 *.frm* 文件 .
4. 执行 UNLOCK TABLES 来释放第 2 步的读锁。

下面以给 sakila.film 表的 rating 列增加一个常量为例来说明。当前列看起来如下：

143

```
mysql> SHOW COLUMNS FROM sakila.film LIKE 'rating';
+--------+------------------------------------+------+-----+---------+-------+
| Field  | Type                               | Null | Key | Default | Extra |
+--------+------------------------------------+------+-----+---------+-------+
| rating | enum('G','PG','PG-13','R','NC-17') | YES  |     | G       |       |
+--------+------------------------------------+------+-----+---------+-------+
```

假设我们需要为那些对电影更加谨慎的父母们增加一个 PG-14 的电影分级：

```
mysql> CREATE TABLE sakila.film_new LIKE sakila.film;
mysql> ALTER TABLE sakila.film_new
    -> MODIFY COLUMN rating ENUM('G','PG','PG-13','R','NC-17', 'PG-14')
    -> DEFAULT 'G';
mysql> FLUSH TABLES WITH READ LOCK;
```

注意，我们是在常量列表的末尾增加一个新的值。如果把新增的值放在中间，例如 PG-13 之后，则会导致已经存在的数据的含义被改变：已经存在的 R 值将变成 PG-14，而已经存在的 NC-17 将成为 R，等等。

接下来用操作系统的命令交换 *.frm* 文件：

```
/var/lib/mysql/sakila# mv film.frm film_tmp.frm
/var/lib/mysql/sakila# mv film_new.frm film.frm
/var/lib/mysql/sakila# mv film_tmp.frm film_new.frm
```

再回到 MySQL 命令行，现在可以解锁表并且看到变更后的效果了：

```
mysql> UNLOCK TABLES;
mysql> SHOW COLUMNS FROM sakila.film LIKE 'rating'\G
*************************** 1. row ***************************
Field: rating
 Type: enum('G','PG','PG-13','R','NC-17','PG-14')
```

最后需要做的是删除为完成这个操作而创建的辅助表：

```
mysql> DROP TABLE sakila.film_new;
```

4.5.2 快速创建 MyISAM 索引

为了高效地载入数据到 MyISAM 表中，有一个常用的技巧是先禁用索引、载入数据，然后重新启用索引：

```
mysql> ALTER TABLE test.load_data DISABLE KEYS;
-- load the data
mysql> ALTER TABLE test.load_data ENABLE KEYS;
```

这个技巧能够发挥作用，是因为构建索引的工作被延迟到数据完全载入以后，这个时候已经可以通过排序来构建索引了。这样做会快很多，并且使得索引树[注17]的碎片更少、更紧凑。

不幸的是，这个办法对唯一索引无效，因为 DISABLE KEYS 只对非唯一索引有效。MyISAM 会在内存中构造唯一索引，并且为载入的每一行检查唯一性。一旦索引的大小 144
超过了有效内存大小，载入操作就会变得越来越慢。

在现代版本的 InnoDB 版本中，有一个类似的技巧，这依赖于 InnoDB 的快速在线索引创建功能。这个技巧是，先删除所有的非唯一索引，然后增加新的列，最后重新创建删除掉的索引。Percona Server 可以自动完成这些操作步骤。

也可以使用像前面说的 ALTER TABLE 的骇客方法来加速这个操作，但需要多做一些工作并且承担一定的风险。这对从备份中载入数据是很有用的，例如，当已经知道所有数据都是有效的并且没有必要做唯一性检查时就可以这么来操作。

> 再次说明，这是没有文档说明并且不受官方支持的技巧。若使用的话，需要自己承担风险，并且操作之前一定要先备份数据。

下面是操作步骤：

1. 用需要的表结构创建一张表，但是不包括索引。

注 17：如果使用的是 LOAD DATA FILE，并且要载入的表是空的，MyISAM 也可以通过排序来构造索引。

2. 载入数据到表中以构建 *.MYD* 文件。
3. 按照需要的结构创建另外一张空表，这次要包含索引。这会创建需要的 *.frm* 和 *.MYI* 文件。
4. 获取读锁并刷新表。
5. 重命名第二张表的 *.frm* 和 *.MYI* 文件，让 MySQL 认为是第一张表的文件。
6. 释放读锁。
7. 使用 REPAIR TABLE 来重建表的索引。该操作会通过排序来构建所有索引，包括唯一索引。

这个操作步骤对大表来说会快很多。

145 4.6 总结

良好的 schema 设计原则是普遍适用的，但 MySQL 有它自己的实现细节要注意。概括来说，尽可能保持任何东西小而简单总是好的。MySQL 喜欢简单，需要使用数据库的人应该也同样会喜欢简单的原则：

- 尽量避免过度设计，例如会导致极其复杂查询的 schema 设计，或者有很多列的表设计（很多的意思是介于有点多和非常多之间）。
- 使用小而简单的合适数据类型，除非真实数据模型中有确切的需要，否则应该尽可能地避免使用 NULL 值。
- 尽量使用相同的数据类型存储相似或相关的值，尤其是要在关联条件中使用的列。
- 注意可变长字符串，其在临时表和排序时可能导致悲观的按最大长度分配内存。
- 尽量使用整型定义标识列。
- 避免使用 MySQL 已经遗弃的特性，例如指定浮点数的精度，或者整数的显示宽度。
- 小心使用 ENUM 和 SET。虽然它们用起来很方便，但是不要滥用，否则有时候会变成陷阱。最好避免使用 BIT。

范式是好的，但是反范式（大多数情况下意味着重复数据）有时也是必需的，并且能带来好处。第 5 章我们将看到更多的例子。预先计算、缓存或生成汇总表也可能获得很大的好处。Justin Swanhart 的 Flexviews 工具可以帮助维护汇总表。

最后，ALTER TABLE 是让人痛苦的操作，因为在大部分情况下，它都会锁表并且重建整张表。我们展示了一些特殊的场景可以使用骇客方法；但是对大部分场景，必须使用其他更常规的方法，例如在备机执行 ALTER 并在完成后把它切换为主库。本书后续章节会有更多关于这方面的内容。

第 5 章 创建高性能的索引

索引（在 MySQL 中也叫做“键（key)”）是存储引擎用于快速找到记录的一种数据结构。这是索引的基本功能，除此之外，本章还将讨论索引其他一些方面有用的属性。

索引对于良好的性能非常关键。尤其是当表中的数据量越来越大时，索引对性能的影响愈发重要。在数据量较小且负载较低时，不恰当的索引对性能的影响可能还不明显，但当数据量逐渐增大时，性能则会急剧下降[注1]。

不过，索引却经常被忽略，有时候甚至被误解，所以在实际案例中经常会遇到由糟糕索引导致的问题。这也是我们把索引优化放在了靠前的章节，甚至比查询优化还靠前的原因。

索引优化应该是对查询性能优化最有效的手段了。索引能够轻易将查询性能提高几个数量级，“最优”的索引有时比一个“好的”索引性能要好两个数量级。创建一个真正“最优”的索引经常需要重写查询，所以，本章和下一章的关系非常紧密。

5.1 索引基础

要理解 MySQL 中索引是如何工作的，最简单的方法就是去看看一本书的“索引”部分：如果想在一本书中找到某个特定主题，一般会先看书的“索引”，找到对应的页码。

在 MySQL 中，存储引擎用类似的方法使用索引，其先在索引中找到对应值，然后根据匹配的索引记录找到对应的数据行。假如要运行下面的查询：

注 1： 除非特别说明，本章假设使用的都是传统的硬盘驱动器。固态硬盘驱动器有着完全不同的性能特性，本书将对此进行详细的描述。然而即使是固态硬盘，索引的原则依然成立，只是那些需要尽量避免的糟糕索引对于固态硬盘的影响没有传统硬盘那么糟糕。

```
mysql> SELECT first_name FROM sakila.actor WHERE actor_id = 5;
```

148 如果在 actor_id 列上建有索引，则 MySQL 将使用该索引找到 actor_id 为 5 的行，也就是说，MySQL 先在索引上按值进行查找，然后返回所有包含该值的数据行。

索引可以包含一个或多个列的值。如果索引包含多个列，那么列的顺序也十分重要，因为 MySQL 只能高效地使用索引的最左前缀列。创建一个包含两个列的索引，和创建两个只包含一列的索引是大不相同的，下面将详细介绍。

如果使用的是 ORM，是否还需要关心索引？

简而言之：是的，仍然需要理解索引，即使是使用对象关系映射（ORM）工具。

ORM 工具能够生产符合逻辑的、合法的查询（多数时候），除非只是生成非常基本的查询（例如仅是根据主键查询），否则它很难生成适合索引的查询。无论是多么复杂的 ORM 工具，在精妙和复杂的索引面前都是“浮云”。读完本章后面的内容以后，你就会同意这个观点的！很多时候，即使是查询优化技术专家也很难兼顾到各种情况，更别说 ORM 了。

5.1.1 索引的类型

索引有很多种类型，可以为不同的场景提供更好的性能。在 MySQL 中，索引是在存储引擎层而不是服务器层实现的。所以，并没有统一的索引标准：不同存储引擎的索引的工作方式并不一样，也不是所有的存储引擎都支持所有类型的索引。即使多个存储引擎支持同一种类型的索引，其底层的实现也可能不同。

下面我们先来看看 MySQL 支持的索引类型，以及它们的优点和缺点。

B-Tree 索引

当人们谈论索引的时候，如果没有特别指明类型，那多半说的是 B-Tree 索引，它使用 B-Tree 数据结构来存储数据[注2]。大多数 MySQL 引擎都支持这种索引。Archive 引擎是一个例外：5.1 之前 Archive 不支持任何索引，直到 5.1 才开始支持单个自增列（AUTO_INCREMENT）的索引。

我们使用术语“B-Tree”，是因为 MySQL 在 CREATE TABLE 和其他语句中也使用该关键字。

注 2： 实际上很多存储引擎使用的是 B+Tree，即每一个叶子节点都包含指向下一个叶子节点的指针，从而方便叶子节点的范围遍历。对于 B-Tree 更详细的细节可以参考相关计算机科学方面的书籍。

不过，底层的存储引擎也可能使用不同的存储结构，例如，NDB 集群存储引擎内部实际上使用了 T-Tree 结构存储这种索引，即使其名字是 BTREE；InnoDB 则使用的是 B+Tree，各种数据结构和算法的变种不在本书的讨论范围之内。 149

存储引擎以不同的方式使用 B-Tree 索引，性能也各有不同，各有优劣。例如，MyISAM 使用前缀压缩技术使得索引更小，但 InnoDB 则按照原数据格式进行存储。再如 MyISAM 索引通过数据的物理位置引用被索引的行，而 InnoDB 则根据主键引用被索引的行。

B-Tree 通常意味着所有的值都是按顺序存储的，并且每一个叶子页到根的距离相同。图 5-1 展示了 B-Tree 索引的抽象表示，大致反映了 InnoDB 索引是如何工作的。MyISAM 使用的结构有所不同，但基本思想是类似的。

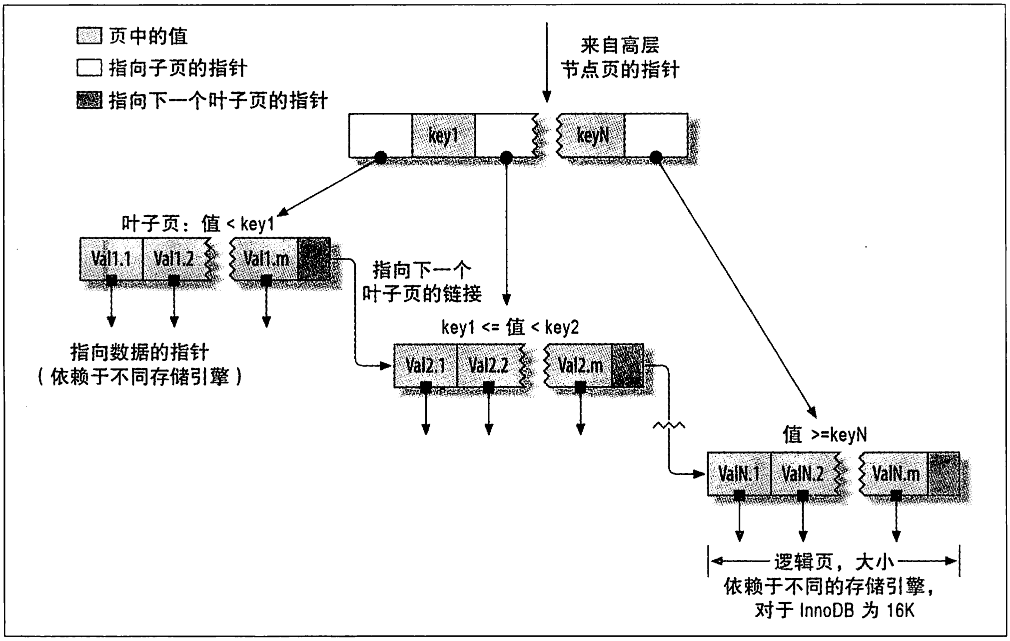


图5-1：建立在B-Tree结构（从技术上来说是B+Tree）上的索引

B-Tree 索引能够加快访问数据的速度，因为存储引擎不再需要进行全表扫描来获取需要的数据，取而代之的是从索引的根节点（图示并未画出）开始进行搜索。根节点的槽中存放了指向子节点的指针，存储引擎根据这些指针向下层查找。通过比较节点页的值和要查找的值可以找到合适的指针进入下层子节点，这些指针实际上定义了子节点页中值的上限和下限。最终存储引擎要么是找到对应的值，要么该记录不存在。

150 叶子节点比较特别，它们的指针指向的是被索引的数据，而不是其他的节点页（不同引擎的“指针”类型不同）。图 5-1 中仅绘制了一个节点和其对应的叶子节点，其实在根节点和叶子节点之间可能有很多层节点页。树的深度和表的大小直接相关。

B-Tree 对索引列是顺序组织存储的，所以很适合查找范围数据。例如，在一个基于文本域的索引树上，按字母顺序传递连续的值进行查找是非常合适的，所以像“找出所有以 I 到 K 开头的名字”这样的查找效率会非常高。

假设有如下数据表：

```
CREATE TABLE People (
   last_name  varchar(50)    not null,
   first_name varchar(50)    not null,
   dob        date           not null,
   gender     enum('m', 'f')not null,
   key(last_name, first_name, dob)
);
```

对于表中的每一行数据，索引中包含了 last_name、first_name 和 dob 列的值，图 5-2 显示了该索引是如何组织数据的存储的。

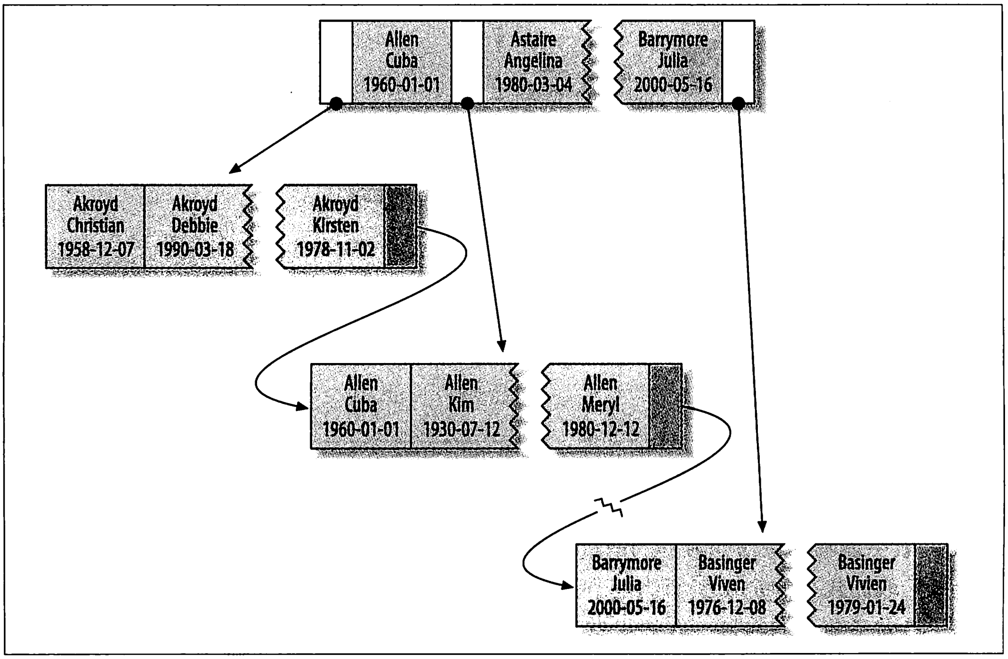


图5-2：B-Tree（从技术上来说是B+Tree）索引树中的部分条目示例

请注意，索引对多个值进行排序的依据是 CREATE TABLE 语句中定义索引时列的顺序。看一下最后两个条目，两个人的姓和名都一样，则根据他们的出生日期来排列顺序。 151

可以使用 B-Tree 索引的查询类型。 B-Tree 索引适用于全键值、键值范围或键前缀查找。其中键前缀查找只适用于根据最左前缀的查找[注3]。前面所述的索引对如下类型的查询有效。

全值匹配

全值匹配指的是和索引中的所有列进行匹配，例如前面提到的索引可用于查找姓名为 Cuba Allen、出生于 1960-01-01 的人。

匹配最左前缀

前面提到的索引可用于查找所有姓为 Allen 的人，即只使用索引的第一列。

匹配列前缀

也可以只匹配某一列的值的开头部分。例如前面提到的索引可用于查找所有以 J 开头的姓的人。这里也只使用了索引的第一列。

匹配范围值

例如前面提到的索引可用于查找姓在 Allen 和 Barrymore 之间的人。这里也只使用了索引的第一列。

精确匹配某一列并范围匹配另外一列

前面提到的索引也可用于查找所有姓为 Allen，并且名字是字母 K 开头（比如 Kim、Karl 等）的人。即第一列 last_name 全匹配，第二列 first_name 范围匹配。

只访问索引的查询

B-Tree 通常可以支持"只访问索引的查询"，即查询只需要访问索引，而无须访问数据行。后面我们将单独讨论这种"覆盖索引"的优化。

因为索引树中的节点是有序的，所以除了按值查找之外，索引还可以用于查询中的 ORDER BY 操作（按顺序查找）。一般来说，如果 B-Tree 可以按照某种方式查找到值，那么也可以按照这种方式用于排序。所以，如果 ORDER BY 子句满足前面列出的几种查询类型，则这个索引也可以满足对应的排序需求。

下面是一些关于 B-Tree 索引的限制：

- 如果不是按照索引的最左列开始查找，则无法使用索引。例如上面例子中的索引无法用于查找名字为 Bill 的人，也无法查找某个特定生日的人，因为这两列都不是最左数据列。类似地，也无法查找姓氏以某个字母结尾的人。 152

注 3： 这是 MySQL 相关的特性，甚至和具体的版本也相关。其他有些数据库也可以使用索引的非前缀部分，虽然使用完全的前缀的效率会更好。MySQL 未来也可能会提供这个特性；本章后面也会介绍一些绕过限制的方法。

- 不能跳过索引中的列。也就是说，前面所述的索引无法用于查找姓为 Smith 并且在某个特定日期出生的人。如果不指定名（first_name），则 MySQL 只能使用索引的第一列。
- 如果查询中有某个列的范围查询，则其右边所有列都无法使用索引优化查找。例如有查询 WHERE last_name='Smith' AND first_name LIKE 'J%' AND dob = '1976-12-23'，这个查询只能使用索引的前两列，因为这里 LIKE 是一个范围条件（但是服务器可以把其余列用于其他目的）。如果范围查询列值的数量有限，那么可以通过使用多个等于条件来代替范围条件。在本章的索引案例学习部分，我们将演示一个详细的案例。

到这里读者应该可以明白，前面提到的索引列的顺序是多么的重要：这些限制都和索引列的顺序有关。在优化性能的时候，可能需要使用相同的列但顺序不同的索引来满足不同类型的查询需求。

也有些限制并不是 B-Tree 本身导致的，而是 MySQL 优化器和存储引擎使用索引的方式导致的，这部分限制在未来的版本中可能就不再是限制了。

哈希索引

哈希索引（hash index）基于哈希表实现，只有精确匹配索引所有列的查询才有效[注4]。对于每一行数据，存储引擎都会对所有的索引列计算一个哈希码（hash code），哈希码是一个较小的值，并且不同键值的行计算出来的哈希码也不一样。哈希索引将所有的哈希码存储在索引中，同时在哈希表中保存指向每个数据行的指针。

在 MySQL 中，只有 Memory 引擎显式支持哈希索引。这也是 Memory 引擎表的默认索引类型，Memory 引擎同时也支持 B-Tree 索引。值得一提的是，Memory 引擎是支持非唯一哈希索引的，这在数据库世界里面是比较与众不同的。如果多个列的哈希值相同，索引会以链表的方式存放多个记录指针到同一个哈希条目中。

下面来看一个例子。假设有如下表：

153

```
CREATE TABLE testhash (
   fname VARCHAR(50) NOT NULL,
   lname VARCHAR(50) NOT NULL,
   KEY USING HASH(fname)
) ENGINE=MEMORY;
```

注 4： 关于哈希表请参考相关计算机科学方面的书籍。

表中包含如下数据：

```
mysql> SELECT * FROM testhash;
+--------+-----------+
| fname  | lname     |
+--------+-----------+
| Arjen  | Lentz     |
| Baron  | Schwartz  |
| Peter  | Zaitsev   |
| Vadim  | Tkachenko |
+--------+-----------+
```

假设索引使用假想的哈希函数 f()，它返回下面的值（都是示例数据，非真实数据）：

```
f('Arjen')= 2323
f('Baron')= 7437
f('Peter')= 8784
f('Vadim')= 2458
```

则哈希索引的数据结构如下：

槽（Slot）	值（Value）
2323	指向第 1 行的指针
2458	指向第 4 行的指针
7437	指向第 2 行的指针
8784	指向第 3 行的指针

注意每个槽的编号是顺序的，但是数据行不是。现在，来看如下查询：

```
mysql> SELECT lname FROM testhash WHERE fname='Peter';
```

MySQL 先计算 'Peter' 的哈希值，并使用该值寻找对应的记录指针。因为 f('Peter')= 8784，所以 MySQL 在索引中查找 8784，可以找到指向第 3 行的指针，最后一步是比较第三行的值是否为 'Peter'，以确保就是要查找的行。

因为索引自身只需存储对应的哈希值，所以索引的结构十分紧凑，这也让哈希索引查找的速度非常快。然而，哈希索引也有它的限制：

- 哈希索引只包含哈希值和行指针，而不存储字段值，所以不能使用索引中的值来避免读取行。不过，访问内存中的行的速度很快，所以大部分情况下这一点对性能的影响并不明显。
- 哈希索引数据并不是按照索引值顺序存储的，所以也就无法用于排序。 154
- 哈希索引也不支持部分索引列匹配查找，因为哈希索引始终是使用索引列的全部内容来计算哈希值的。例如，在数据列 (A,B) 上建立哈希索引，如果查询只有数据列 A，则无法使用该索引。

- 哈希索引只支持等值比较查询，包括 =、IN()、<=>（注意 <> 和 <=> 是不同的操作）。也不支持任何范围查询，例如 WHERE price > 100。
- 访问哈希索引的数据非常快，除非有很多哈希冲突（不同的索引列值却有相同的哈希值）。当出现哈希冲突的时候，存储引擎必须遍历链表中所有的行指针，逐行进行比较，直到找到所有符合条件的行。
- 如果哈希冲突很多的话，一些索引维护操作的代价也会很高。例如，如果在某个选择性很低（哈希冲突很多）的列上建立哈希索引，那么当从表中删除一行时，存储引擎需要遍历对应哈希值的链表中的每一行，找到并删除对应行的引用，冲突越多，代价越大。

因为这些限制，哈希索引只适用于某些特定的场合。而一旦适合哈希索引，则它带来的性能提升将非常显著。举个例子，在数据仓库应用中有一种经典的“星型”schema，需要关联很多查找表，哈希索引就非常适合查找表的需求。

除了 Memory 引擎外，NDB 集群引擎也支持唯一哈希索引，且在 NDB 集群引擎中作用非常特殊，但这不属于本书的范围。

InnoDB 引擎有一个特殊的功能叫做“自适应哈希索引（adaptive hash index）”。当 InnoDB 注意到某些索引值被使用得非常频繁时，它会在内存中基于 B-Tree 索引之上再创建一个哈希索引，这样就让 B-Tree 索引也具有哈希索引的一些优点，比如快速的哈希查找。这是一个完全自动的、内部的行为，用户无法控制或者配置，不过如果有必要，完全可以关闭该功能。

创建自定义哈希索引。如果存储引擎不支持哈希索引，则可以模拟像 InnoDB 一样创建哈希索引，这可以享受一些哈希索引的便利，例如只需要很小的索引就可以为超长的键创建索引。

思路很简单：在 B-Tree 基础上创建一个伪哈希索引。这和真正的哈希索引不是一回事，因为还是使用 B-Tree 进行查找，但是它使用哈希值而不是键本身进行索引查找。你需要做的就是在查询的 WHERE 子句中手动指定使用哈希函数。

155 下面是一个实例，例如需要存储大量的 URL，并需要根据 URL 进行搜索查找。如果使用 B-Tree 来存储 URL，存储的内容就会很大，因为 URL 本身都很长。正常情况下会有如下查询：

```
mysql> SELECT id FROM url WHERE url="http://www.mysql.com";
```

若删除原来 URL 列上的索引，而新增一个被索引的 url_crc 列，使用 CRC32 做哈希，就可以使用下面的方式查询：

```
mysql> SELECT id FROM url WHERE url="http://www.mysql.com"
    ->    AND url_crc=CRC32("http://www.mysql.com");
```

这样做的性能会非常高，因为 MySQL 优化器会使用这个选择性很高而体积很小的基于 url_crc 列的索引来完成查找（在上面的案例中，索引值为 1560514994）。即使有多个记录有相同的索引值，查找仍然很快，只需要根据哈希值做快速的整数比较就能找到索引条目，然后一一比较返回对应的行。另外一种方式就是对完整的 URL 字符串做索引，那样会非常慢。

这样实现的缺陷是需要维护哈希值。可以手动维护，也可以使用触发器实现。下面的案例演示了触发器如何在插入和更新时维护 url_crc 列。首先创建如下表：

```
CREATE TABLE pseudohash (
   id int unsigned NOT NULL auto_increment,
   url varchar(255) NOT NULL,
   url_crc int unsigned NOT NULL DEFAULT 0,
   PRIMARY KEY(id)
);
```

然后创建触发器。先临时修改一下语句分隔符，这样就可以在触发器定义中使用分号：

```
DELIMITER //

CREATE TRIGGER pseudohash_crc_ins BEFORE INSERT ON pseudohash FOR EACH ROW BEGIN
SET NEW.url_crc=crc32(NEW.url);
END;
//

CREATE TRIGGER pseudohash_crc_upd BEFORE UPDATE ON pseudohash FOR EACH ROW BEGIN
SET NEW.url_crc=crc32(NEW.url);
END;
//

DELIMITER ;
```

剩下的工作就是验证一下触发器如何维护哈希索引： 156

```
mysql> INSERT INTO pseudohash (url) VALUES ('http://www.mysql.com');
mysql> SELECT * FROM pseudohash;
+----+----------------------+------------+
| id | url                  | url_crc    |
+----+----------------------+------------+
|  1 | http://www.mysql.com | 1560514994 |
+----+----------------------+------------+
mysql> UPDATE pseudohash SET url='http://www.mysql.com/' WHERE id=1;
mysql> SELECT * FROM pseudohash;
+----+---------------------+ +------------+
| id | url                   | url_crc    |
+----+---------------------+ +------------+
|  1 | http://www.mysql.com/ | 1558250469 |
+----+---------------------+ +------------+
```

如果采用这种方式，记住不要使用 SHA1() 和 MD5() 作为哈希函数。因为这两个函数计算出来的哈希值是非常长的字符串，会浪费大量空间，比较时也会更慢。SHA1() 和 MD5() 是强加密函数，设计目标是最大限度消除冲突，但这里并不需要这样高的要求。简单哈希函数的冲突在一个可以接受的范围，同时又能够提供更好的性能。

如果数据表非常大，CRC32() 会出现大量的哈希冲突，则可以考虑自己实现一个简单的 64 位哈希函数。这个自定义函数要返回整数，而不是字符串。一个简单的办法可以使用 MD5() 函数返回值的一部分来作为自定义哈希函数。这可能比自己写一个哈希算法的性能要差（参考第 7 章），不过这样实现最简单：

```
mysql> SELECT CONV(RIGHT(MD5('http://www.mysql.com/'), 16), 16, 10) AS HASH64;
+----------------------+
| HASH64               |
+----------------------+
| 9761173720318281581 |
+----------------------+
```

处理哈希冲突。当使用哈希索引进行查询的时候，必须在 WHERE 子句中包含常量值：

```
mysql> SELECT id FROM url WHERE url_crc=CRC32("http://www.mysql.com")
    ->    AND url="http://www.mysql.com";
```

一旦出现哈希冲突，另一个字符串的哈希值也恰好是 1560514994，则下面的查询是无法正确工作的。

```
mysql> SELECT id FROM url WHERE url_crc=CRC32("http://www.mysql.com");
```

因为所谓的“生日悖论”[注5]，出现哈希冲突的概率的增长速度可能比想象的要快得多。CRC32() 返回的是 32 位的整数，当索引有 93 000 条记录时出现冲突的概率是 1%。例如我们将 */usr/share/dict/words* 中的词导入数据表并进行 CRC32() 计算，最后会有 98 569 行。这就已经出现一次哈希冲突了，冲突让下面的查询返回了多条记录：

157

```
mysql> SELECT word, crc FROM words WHERE crc = CRC32('gnu');
+---------+------------+
| word    | crc        |
+---------+------------+
| codding | 1774765869 |
| gnu     | 1774765869 |
+---------+------------+
```

注 5： 参考 *http://en.wikipedia.org/wiki/Birthday_problem*。——译者注

正确的写法应该如下：

```
mysql> SELECT word, crc FROM words WHERE crc = CRC32('gnu')AND word = 'gnu';
+------+------------+
| word | crc        |
+------+------------+
| gnu  | 1774765869 |
+------+------------+
```

要避免冲突问题，必须在 WHERE 条件中带入哈希值和对应列值。如果不是想查询具体值，例如只是统计记录数（不精确的），则可以不带入列值，直接使用 CRC32() 的哈希值查询即可。还可以使用如 FNV64() 函数作为哈希函数，这是移植自 Percona Server 的函数，可以以插件的方式在任何 MySQL 版本中使用，哈希值为 64 位，速度快，且冲突比 CRC32() 要少很多。

空间数据索引 (R-Tree)

MyISAM 表支持空间索引，可以用作地理数据存储。和 B-Tree 索引不同，这类索引无须前缀查询。空间索引会从所有维度来索引数据。查询时，可以有效地使用任意维度来组合查询。必须使用 MySQL 的 GIS 相关函数如 MBRCONTAINS() 等来维护数据。MySQL 的 GIS 支持并不完善，所以大部分人都不会使用这个特性。开源关系数据库系统中对 GIS 的解决方案做得比较好的是 PostgreSQL 的 PostGIS。

全文索引

全文索引是一种特殊类型的索引，它查找的是文本中的关键词，而不是直接比较索引中的值。全文搜索和其他几类索引的匹配方式完全不一样。它有许多需要注意的细节，如停用词、词干和复数、布尔搜索等。全文索引更类似于搜索引擎做的事情，而不是简单的 WHERE 条件匹配。

在相同的列上同时创建全文索引和基于值的 B-Tree 索引不会有冲突，全文索引适用于 MATCH AGAINST 操作，而不是普通的 WHERE 条件操作。

我们将在第 7 章讨论更多的全文索引的细节。

其他索引类别

158

还有很多第三方的存储引擎使用不同类型的数据结构来存储索引。例如 TokuDB 使用分形树索引（fractal tree index），这是一类较新开发的数据结构，既有 B-Tree 的很多优点，也避免了 B-Tree 的一些缺点。如果通读完本章，可以看到很多关于 InnoDB 的主题，包括聚簇索引、覆盖索引等。多数情况下，针对 InnoDB 的讨论也都适用于 TokuDB。

ScaleDB 使用 Patricia tries（这个词不是拼写错误），其他一些存储引擎技术如 InfiniDB 和 Infobright 则使用了一些特殊的数据结构来优化某些特殊的查询。

5.2 索引的优点

索引可以让服务器快速地定位到表的指定位置。但是这并不是索引的唯一作用，到目前为止可以看到，根据创建索引的数据结构不同，索引也有一些其他的附加作用。

最常见的 B-Tree 索引，按照顺序存储数据，所以 MySQL 可以用来做 `ORDER BY` 和 `GROUP BY` 操作。因为数据是有序的，所以 B-Tree 也就会将相关的列值都存储在一起。最后，因为索引中存储了实际的列值，所以某些查询只使用索引就能够完成全部查询。据此特性，总结下来索引有如下三个优点：

1. 索引大大减少了服务器需要扫描的数据量。
2. 索引可以帮助服务器避免排序和临时表。
3. 索引可以将随机 I/O 变为顺序 I/O。

“索引”这个主题完全值得单独写一本书，如果想深入理解这部分内容，强烈建议阅读由 Tapio Lahdenmaki 和 Mike Leach 编写的 *Relational Database Index Design and the Optimizers*（Wiley 出版社）一书，该书详细介绍了如何计算索引的成本和作用、如何评估查询速度、如何分析索引维护的代价和其带来的好处等。

Lahdenmaki 和 Leach 在书中介绍了如何评价一个索引是否适合某个查询的“三星系统”（three-star system）：索引将相关的记录放到一起则获得一星；如果索引中的数据顺序和查找中的排列顺序一致则获得二星；如果索引中的列包含了查询中需要的全部列则获得“三星”。后面我们将会介绍这些原则。

159

索引是最好的解决方案吗？

索引并不总是最好的工具。总的来说，只有当索引帮助存储引擎快速查找到记录带来的好处大于其带来的额外工作时，索引才是有效的。对于非常小的表，大部分情况下简单的全表扫描更高效。对于中到大型的表，索引就非常有效。但对于特大型的表，建立和使用索引的代价将随之增长。这种情况下，则需要一种技术可以直接区分出查询需要的一组数据，而不是一条记录一条记录地匹配。例如可以使用分区技术，请参考第 7 章。

如果表的数量特别多，可以建立一个元数据信息表，用来查询需要用到的某些特性。例如执行那些需要聚合多个应用分布在多个表的数据的查询，则需要记录“哪个用户的信息存储在哪个表中”的元数据，这样在查询时就可以直接忽略那些不包含指定用户信息的表。对于大型系统，这是一个常用的技巧。事实上，Infobright就是使用类似的实现。对于TB级别的数据，定位单条记录的意义不大，所以经常会使用块级别元数据技术来替代索引。

5.3 高性能的索引策略

正确地创建和使用索引是实现高性能查询的基础。前面已经介绍了各种类型的索引及其对应的优缺点。现在我们一起来看看如何真正地发挥这些索引的优势。

高效地选择和使用索引有很多种方式，其中有些是针对特殊案例的优化方法，有些则是针对特定行为的优化。使用哪个索引，以及如何评估选择不同索引的性能影响的技巧，则需要持续不断地学习。接下来的几个小节将帮助读者理解如何高效地使用索引。

5.3.1 独立的列

我们通常会看到一些查询不当地使用索引，或者使得MySQL无法使用已有的索引。如果查询中的列不是独立的，则MySQL就不会使用索引。“独立的列”是指索引列不能是表达式的一部分，也不能是函数的参数。

例如，下面这个查询无法使用actor_id列的索引：

```
mysql> SELECT actor_id FROM sakila.actor WHERE actor_id + 1 = 5;
```

凭肉眼很容易看出WHERE中的表达式其实等价于actor_id = 4，但是MySQL无法自动解析这个方程式。这完全是用户行为。我们应该养成简化WHERE条件的习惯，始终将索引列单独放在比较符号的一侧。

下面是另一个常见的错误： 160

```
mysql> SELECT ... WHERE TO_DAYS(CURRENT_DATE) - TO_DAYS(date_col) <= 10;
```

5.3.2 前缀索引和索引选择性

有时候需要索引很长的字符列，这会让索引变得大且慢。一个策略是前面提到过的模拟哈希索引。但有时候这样做还不够，还可以做些什么呢？

通常可以索引开始的部分字符，这样可以大大节约索引空间，从而提高索引效率。但这样也会降低索引的选择性。索引的选择性是指，不重复的索引值（也称为基数，cardinality）和数据表的记录总数（*#T*）的比值，范围从 *1/#T* 到 1 之间。索引的选择性越高则查询效率越高，因为选择性高的索引可以让 MySQL 在查找时过滤掉更多的行。唯一索引的选择性是 1，这是最好的索引选择性，性能也是最好的。

一般情况下某个列前缀的选择性也是足够高的，足以满足查询性能。对于 BLOB、TEXT 或者很长的 VARCHAR 类型的列，必须使用前缀索引，因为 MySQL 不允许索引这些列的完整长度。

诀窍在于要选择足够长的前缀以保证较高的选择性，同时又不能太长（以便节约空间）。前缀应该足够长，以使得前缀索引的选择性接近于索引整个列。换句话说，前缀的“基数”应该接近于完整列的“基数”。

为了决定前缀的合适长度，需要找到最常见的值的列表，然后和最常见的前缀列表进行比较。在示例数据库 Sakila 中并没有合适的例子，所以我们从表 city 中生成一个示例表，这样就有足够的数据进行演示：

```
CREATE TABLE sakila.city_demo(city VARCHAR(50) NOT NULL);
INSERT INTO sakila.city_demo(city) SELECT city FROM sakila.city;
-- Repeat the next statement five times:
INSERT INTO sakila.city_demo(city) SELECT city FROM sakila.city_demo;
-- Now randomize the distribution (inefficiently but conveniently):
UPDATE sakila.city_demo
   SET city = (SELECT city FROM  sakila.city ORDER BY RAND() LIMIT 1);
```

现在我们有了示例数据集。数据分布当然不是真实的分布；因为我们使用了 RAND()，所以你的结果会与此不同，但对这个练习来说这并不重要。首先，我们找到最常见的城市列表：

```
mysql> SELECT COUNT(*) AS cnt, city
    -> FROM sakila.city_demo GROUP BY city ORDER BY cnt DESC LIMIT 10;
+-----+----------------+
| cnt | city           |
+-----+----------------+
|  65 | London         |
|  49 | Hiroshima      |
|  48 | Teboksary      |
|  48 | Pak Kret       |
|  48 | Yaound         |
|  47 | Tel Aviv-Jaffa |
|  47 | Shimoga        |
|  45 | Cabuyao        |
|  45 | Callao         |
|  45 | Bislig         |
+-----+----------------+
```

161

注意到，上面每个值都出现了 45 ～ 65 次。现在查找到最频繁出现的城市前缀，先从 3 个前缀字母开始：

```
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 3) AS pref
    -> FROM sakila.city_demo GROUP BY pref ORDER BY cnt DESC LIMIT 10;
+-----+------+
| cnt | pref |
+-----+------+
| 483 | San  |
| 195 | Cha  |
| 177 | Tan  |
| 167 | Sou  |
| 163 | al-  |
| 163 | Sal  |
| 146 | Shi  |
| 136 | Hal  |
| 130 | Val  |
| 129 | Bat  |
+-----+------+
```

每个前缀都比原来的城市出现的次数更多，因此唯一前缀比唯一城市要少得多。然后我们增加前缀长度，直到这个前缀的选择性接近完整列的选择性。经过实验后发现前缀长度为 7 时比较合适：

```
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 7) AS pref
    -> FROM sakila.city_demo GROUP BY pref ORDER BY cnt DESC LIMIT 10;
+-----+---------+
| cnt | pref    |
+-----+---------+
|  70 | Santiag |
|  68 | San Fel |
|  65 | London  |
|  61 | Valle d |
|  49 | Hiroshi |
|  48 | Teboksa |
|  48 | Pak Kre |
|  48 | Yaound  |
|  47 | Tel Avi |
|  47 | Shimoga |
+-----+---------+
```

计算合适的前缀长度的另外一个办法就是计算完整列的选择性，并使前缀的选择性接近于完整列的选择性。下面显示如何计算完整列的选择性：

```
mysql> SELECT COUNT(DISTINCT city)/COUNT(*) FROM sakila.city_demo;
+-------------------------------+
| COUNT(DISTINCT city)/COUNT(*) |
+-------------------------------+
|                        0.0312 |
+-------------------------------+
```

162

通常来说（尽管也有例外情况），这个例子中如果前缀的选择性能够接近 0.031，基本上就可用了。可以在一个查询中针对不同前缀长度进行计算，这对于大表非常有用。下面给出了如何在同一个查询中计算不同前缀长度的选择性：

```
mysql> SELECT COUNT(DISTINCT LEFT(city, 3))/COUNT(*) AS sel3,
    ->    COUNT(DISTINCT LEFT(city, 4))/COUNT(*) AS sel4,
    ->    COUNT(DISTINCT LEFT(city, 5))/COUNT(*) AS sel5,
    ->    COUNT(DISTINCT LEFT(city, 6))/COUNT(*) AS sel6,
    ->    COUNT(DISTINCT LEFT(city, 7))/COUNT(*) AS sel7
    -> FROM sakila.city_demo;
+--------+--------+--------+--------+--------+
| sel3   | sel4   | sel5   | sel6   | sel7   |
+--------+--------+--------+--------+--------+
| 0.0239 | 0.0293 | 0.0305 | 0.0309 | 0.0310 |
+--------+--------+--------+--------+--------+
```

查询显示当前缀长度到达 7 的时候，再增加前缀长度，选择性提升的幅度已经很小了。

只看平均选择性是不够的，也有例外的情况，需要考虑最坏情况下的选择性。平均选择性会让你认为前缀长度为 4 或者 5 的索引已经足够了，但如果数据分布很不均匀，可能就会有陷阱。如果观察前缀为 4 的最常出现城市的次数，可以看到明显不均匀：

```
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 4) AS pref
    -> FROM sakila.city_demo GROUP BY pref ORDER BY cnt DESC LIMIT 5;
+-----+------+
| cnt | pref |
+-----+------+
| 205 | San  |
| 200 | Sant |
| 135 | Sout |
| 104 | Chan |
|  91 | Toul |
+-----+------+
```

如果前缀是 4 个字节，则最常出现的前缀的出现次数比最常出现的城市的出现次数要大很多。即这些值的选择性比平均选择性要低。如果有比这个随机生成的示例更真实的数据，就更有可能看到这种现象。例如在真实的城市名上建一个长度为 4 的前缀索引，对于以“San”和“New”开头的城市的选择性就会非常糟糕，因为很多城市都以这两个词开头。

163 在上面的示例中，已经找到了合适的前缀长度，下面演示一下如何创建前缀索引：

```
mysql> ALTER TABLE sakila.city_demo ADD KEY (city(7));
```

前缀索引是一种能使索引更小、更快的有效办法，但另一方面也有其缺点：MySQL 无法使用前缀索引做 ORDER BY 和 GROUP BY，也无法使用前缀索引做覆盖扫描。

一个常见的场景是针对很长的十六进制唯一 ID 使用前缀索引。在前面的章节中已经讨

论了很多有效的技术来存储这类 ID 信息，但如果使用的是打包过的解决方案，因而无法修改存储结构，那该怎么办？例如使用 vBulletin 或者其他基于 MySQL 的应用在存储网站的会话（SESSION）时，需要在一个很长的十六进制字符串上创建索引。此时如果采用长度为 8 的前缀索引通常能显著地提升性能，并且这种方法对上层应用完全透明。

> 有时候后缀索引（suffix index）也有用途（例如，找到某个域名的所有电子邮件地址）。MySQL 原生并不支持反向索引，但是可以把字符串反转后存储，并基于此建立前缀索引。可以通过触发器来维护这种索引。参考 5.1 节中“创建自定义哈希索引”部分的相关内容。

5.3.3 多列索引

很多人对多列索引的理解都不够。一个常见的错误就是，为每个列创建独立的索引，或者按照错误的顺序创建多列索引。

我们会在 5.3.4 节中单独讨论索引列的顺序问题。先来看第一个问题，为每个列创建独立的索引，从 SHOW CREATE TABLE 中很容易看到这种情况：

```
CREATE TABLE t (
  c1 INT,
  c2 INT,
  c3 INT,
  KEY(c1),
  KEY(c2),
  KEY(c3)
);
```

这种索引策略，一般是由于人们听到一些专家诸如“把 WHERE 条件里面的列都建上索引”这样模糊的建议导致的。实际上这个建议是非常错误的。这样一来最好的情况下也只能是“一星”索引，其性能比起真正最优的索引可能差几个数量级。有时如果无法设计一个“三星”索引，那么不如忽略掉 WHERE 子句，集中精力优化索引列的顺序，或者创建一个全覆盖索引。

在多个列上建立独立的单列索引大部分情况下并不能提高 MySQL 的查询性能。MySQL 5.0 和更新版本引入了一种叫“索引合并”(index merge) 的策略，一定程度上可以使用 164
表上的多个单列索引来定位指定的行。更早版本的 MySQL 只能使用其中某一个单列索引，然而这种情况下没有哪一个独立的单列索引是非常有效的。例如，表 film_actor 在字段 film_id 和 actor_id 上各有一个单列索引。但对于下面这个查询 WHERE 条件，这两个单列索引都不是好的选择：

```
mysql> SELECT film_id, actor_id FROM sakila.film_actor
    -> WHERE actor_id = 1 OR film_id = 1;
```

在老的 MySQL 版本中，MySQL 对这个查询会使用全表扫描。除非改写成如下的两个查询 UNION 的方式：

```
mysql> SELECT film_id, actor_id FROM sakila.film_actor WHERE actor_id = 1
    -> UNION ALL
    -> SELECT film_id, actor_id FROM sakila.film_actor WHERE film_id = 1
    ->    AND actor_id <> 1;
```

但在 MySQL 5.0 和更新的版本中，查询能够同时使用这两个单列索引进行扫描，并将结果进行合并。这种算法有三个变种：OR 条件的联合（union），AND 条件的相交（intersection），组合前两种情况的联合及相交。下面的查询就是使用了两个索引扫描的联合，通过 EXPLAIN 中的 Extra 列可以看到这点：

```
mysql> EXPLAIN SELECT film_id, actor_id FROM sakila.film_actor
    -> WHERE actor_id = 1 OR film_id = 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: index_merge
possible_keys: PRIMARY,idx_fk_film_id
          key: PRIMARY,idx_fk_film_id
      key_len: 2,2
          ref: NULL
         rows: 29
        Extra: Using union(PRIMARY,idx_fk_film_id); Using where
```

MySQL 会使用这类技术优化复杂查询，所以在某些语句的 Extra 列中还可以看到嵌套操作。

索引合并策略有时候是一种优化的结果，但实际上更多时候说明了表上的索引建得很糟糕：

- 当出现服务器对多个索引做相交操作时（通常有多个 AND 条件），通常意味着需要一个包含所有相关列的多列索引，而不是多个独立的单列索引。
- 当服务器需要对多个索引做联合操作时（通常有多个 OR 条件），通常需要耗费大量 CPU 和内存资源在算法的缓存、排序和合并操作上。特别是当其中有些索引的选择性不高，需要合并扫描返回的大量数据的时候。
- 165 更重要的是，优化器不会把这些计算到"查询成本"（cost）中，优化器只关心随机页面读取。这会使得查询的成本被"低估"，导致该执行计划还不如直接走全表扫描。这样做不但会消耗更多的 CPU 和内存资源，还可能会影响查询的并发性，但如果是单独运行这样的查询则往往会忽略对并发性的影响。通常来说，还不如像在 MySQL 4.1 或者更早的时代一样，将查询改写成 UNION 的方式往往更好。

如果在 EXPLAIN 中看到有索引合并，应该好好检查一下查询和表的结构，看是不是已

经是最优的。也可以通过参数 optimizer_switch 来关闭索引合并功能。也可以使用 IGNORE INDEX 提示让优化器忽略掉某些索引。

5.3.4 选择合适的索引列顺序

我们遇到的最容易引起困惑的问题就是索引列的顺序。正确的顺序依赖于使用该索引的查询，并且同时需要考虑如何更好地满足排序和分组的需要（顺便说明，本节内容适用于 B-Tree 索引；哈希或者其他类型的索引并不会像 B-Tree 索引一样按顺序存储数据）。

在一个多列 B-Tree 索引中，索引列的顺序意味着索引首先按照最左列进行排序，其次是第二列，等等。所以，索引可以按照升序或者降序进行扫描，以满足精确符合列顺序的 ORDER BY、GROUP BY 和 DISTINCT 等子句的查询需求。

所以多列索引的列顺序至关重要。在 Lahdenmaki 和 Leach 的"三星索引"系统中，列顺序也决定了一个索引是否能够成为一个真正的"三星索引"（关于三星索引可以参考本章前面的 5.2 节）。在本章的后续部分我们将通过大量的例子来说明这一点。

对于如何选择索引的列顺序有一个经验法则：将选择性最高的列放到索引最前列。这个建议有用吗？在某些场景可能有帮助，但通常不如避免随机 IO 和排序那么重要，考虑问题需要更全面（场景不同则选择不同，没有一个放之四海皆准的法则。这里只是说明，这个经验法则可能没有你想象的重要）。

当不需要考虑排序和分组时，将选择性最高的列放在前面通常是很好的。这时候索引的作用只是用于优化 WHERE 条件的查找。在这种情况下，这样设计的索引确实能够最快地过滤出需要的行，对于在 WHERE 子句中只使用了索引部分前缀列的查询来说选择性也更高。然而，性能不只是依赖于所有索引列的选择性（整体基数），也和查询条件的具体值有关，也就是和值的分布有关。这和前面介绍的选择前缀的长度需要考虑的地方一样。可能需要根据那些运行频率最高的查询来调整索引列的顺序，让这种情况下索引的选择性最高。

以下面的查询为例： 166

```
SELECT * FROM payment WHERE staff_id = 2 AND customer_id = 584;
```

是应该创建一个 (staff_id, customer_id) 索引还是应该颠倒一下顺序？可以跑一些查询来确定在这个表中值的分布情况，并确定哪个列的选择性更高。先用下面的查询预测一下[注6]，看看各个 WHERE 条件的分支对应的数据基数有多大：

注 6： 某些优化极客 (geek) 将这称之为"sarg"，这是"可搜索的参数 (searchable argument)"的缩写。好吧，学会了这个词你也是一个极客了。

```
mysql> SELECT SUM(staff_id = 2), SUM(customer_id = 584) FROM payment\G
*************************** 1. row ***************************
     SUM(staff_id = 2): 7992
SUM(customer_id = 584): 30
```

根据前面的经验法则，应该将索引列 customer_id 放到前面，因为对应条件值的 customer_id 数量更小。我们再来看看对于这个 customer_id 的条件值，对应的 staff_id 列的选择性如何：

```
mysql> SELECT SUM(staff_id = 2) FROM payment WHERE customer_id = 584\G
*************************** 1. row ***************************
SUM(staff_id = 2): 17
```

这样做有一个地方需要注意，查询的结果非常依赖于选定的具体值。如果按上述办法优化，可能对其他一些条件值的查询不公平，服务器的整体性能可能变得更糟，或者其他某些查询的运行变得不如预期。

如果是从诸如 *pt-query-digest* 这样的工具的报告中提取"最差"查询，那么再按上述办法选定的索引顺序往往是非常高效的。如果没有类似的具体查询来运行，那么最好还是按经验法则来做，因为经验法则考虑的是全局基数和选择性，而不是某个具体查询：

```
mysql> SELECT COUNT(DISTINCT staff_id)/COUNT(*) AS staff_id_selectivity,
     > COUNT(DISTINCT customer_id)/COUNT(*) AS customer_id_selectivity,
     > COUNT(*)
     > FROM payment\G
*************************** 1. row ***************************
   staff_id_selectivity: 0.0001
customer_id_selectivity: 0.0373
               COUNT(*): 16049
```

customer_id 的选择性更高，所以答案是将其作为索引列的第一列：

```
mysql> ALTER TABLE payment ADD KEY(customer_id, staff_id);
```

当使用前缀索引的时候，在某些条件值的基数比正常值高的时候，问题就来了。例如，在某些应用程序中，对于没有登录的用户，都将其用户名记录为"guset"，在记录用户行为的会话（session）表和其他记录用户活动的表中"guest"就成为了一个特殊用户
167 ID。一旦查询涉及这个用户，那么和对于正常用户的查询就大不同了，因为通常有很多会话都是没有登录的。系统账号也会导致类似的问题。一个应用通常都有一个特殊的管理员账号，和普通账号不同，它并不是一个具体的用户，系统中所有的其他用户都是这个用户的好友，所以系统往往通过它向网站的所有用户发送状态通知和其他消息。这个账号的巨大的好友列表很容易导致网站出现服务器性能问题。

这实际上是一个非常典型的问题。任何的异常用户，不仅仅是那些用于管理应用的设计糟糕的账号会有同样的问题；那些拥有大量好友、图片、状态、收藏的用户，也会有前

面提到的系统账号同样的问题。

下面是一个我们遇到过的真实案例，在一个用户分享购买商品和购买经验的论坛上，这个特殊表上的查询运行得非常慢：

```
mysql> SELECT COUNT(DISTINCT threadId) AS COUNT_VALUE
    -> FROM Message
    -> WHERE (groupId = 10137) AND (userId = 1288826) AND (anonymous = 0)
    -> ORDER BY priority DESC, modifiedDate DESC
```

这个查询看似没有建立合适的索引，所以客户咨询我们是否可以优化。EXPLAIN的结果如下：

```
           id: 1
  select_type: SIMPLE
        table: Message
         type: ref
          key: ix_groupId_userId
      key_len: 18
          ref: const,const
         rows: 1251162
        Extra: Using where
```

MySQL 为这个查询选择了索引 (groupId, userId)，如果不考虑列的基数，这看起来是一个非常合理的选择。但如果考虑一下 user ID 和 group ID 条件匹配的行数，可能就会有不同的想法了：

```
mysql> SELECT COUNT(*), SUM(groupId = 10137),
    -> SUM(userId = 1288826), SUM(anonymous = 0)
    -> FROM Message\G
*************************** 1. row ***************************
             count(*): 4142217
  sum(groupId = 10137): 4092654
sum(userId = 1288826): 1288496
   sum(anonymous = 0): 4141934
```

从上面的结果来看符合组（groupId）条件几乎满足表中的所有行，符合用户（userId）条件的有 130 万条记录——也就是说索引基本上没什么用。因为这些数据是从其他应用中迁移过来的，迁移的时候把所有的消息都赋予了管理员组的用户。这个案例的解决办法是修改应用程序代码，区分这类特殊用户和组，禁止针对这类用户和组执行这个查询。

从这个小案例可以看到经验法则和推论在多数情况是有用的，但要注意不要假设平均情况下的性能也能代表特殊情况下的性能，特殊情况可能会摧毁整个应用的性能。 168

最后，尽管关于选择性和基数的经验法则值得去研究和分析，但一定要记住别忘了WHERE子句中的排序、分组和范围条件等其他因素，这些因素可能对查询的性能造成非常大的影响。

5.3.5 聚簇索引

聚簇索引[注7]并不是一种单独的索引类型，而是一种数据存储方式。具体的细节依赖于其实现方式，但 InnoDB 的聚簇索引实际上在同一个结构中保存了 B-Tree 索引和数据行。

当表有聚簇索引时，它的数据行实际上存放在索引的叶子页（leaf page）中。术语“聚簇”表示数据行和相邻的键值紧凑地存储在一起[注8]。因为无法同时把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引（不过，覆盖索引可以模拟多个聚簇索引的情况，本章后面将详细介绍）。

因为是存储引擎负责实现索引，因此不是所有的存储引擎都支持聚簇索引。本节我们主要关注 InnoDB，但是这里讨论的原理对于任何支持聚簇索引的存储引擎都是适用的。

图 5-3 展示了聚簇索引中的记录是如何存放的。注意到，叶子页包含了行的全部数据，但是节点页只包含了索引列。在这个案例中，索引列包含的是整数值。

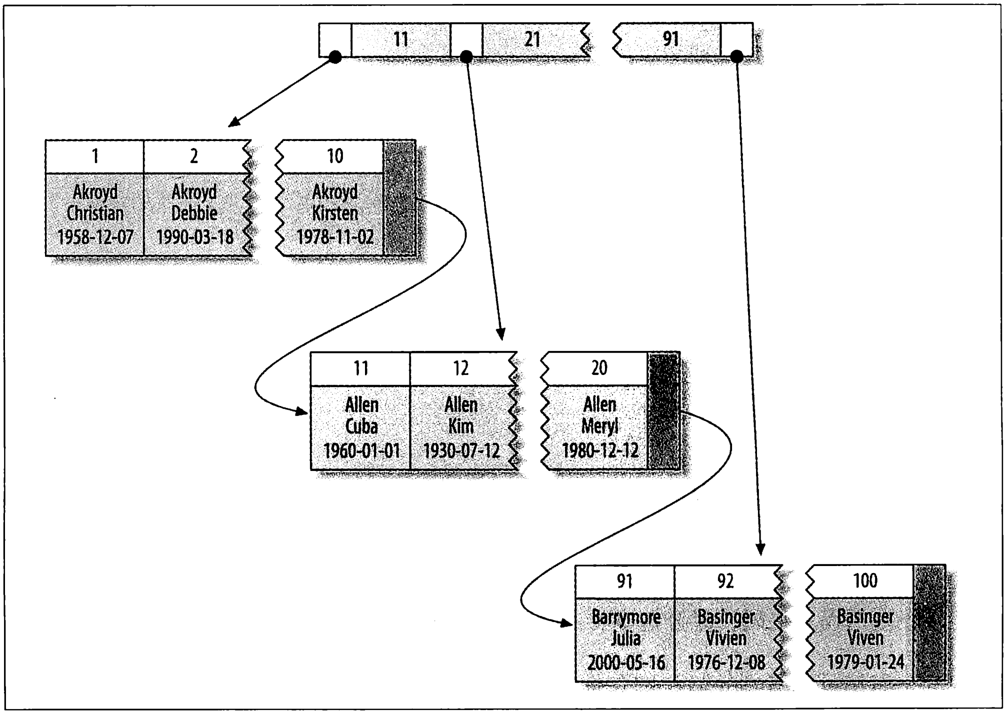


图5-3：聚簇索引的数据分布

注 7： Oracle 用户可能更熟悉索引组织表（index- organized table）的说法，实际上是一样的意思。

注 8： 这并非总成立，很快就可以看到。

一些数据库服务器允许选择哪个索引作为聚簇索引，但直到本书写作之际，还没有任何一个 MySQL 内建的存储引擎支持这一点。InnoDB 将通过主键聚集数据，这也就是说图 5-3 中的“被索引的列”就是主键列。 169

如果没有定义主键，InnoDB 会选择一个唯一的非空索引代替。如果没有这样的索引，InnoDB 会隐式定义一个主键来作为聚簇索引。InnoDB 只聚集在同一个页面中的记录。包含相邻键值的页面可能会相距甚远。

聚簇主键可能对性能有帮助，但也可能导致严重的性能问题。所以需要仔细地考虑聚簇索引，尤其是将表的存储引擎从 InnoDB 改成其他引擎的时候（反过来也一样）。

聚集的数据有一些重要的优点：

- 可以把相关数据保存在一起。例如实现电子邮箱时，可以根据用户 ID 来聚集数据，这样只需要从磁盘读取少数的数据页就能获取某个用户的全部邮件。如果没有使用聚簇索引，则每封邮件都可能导致一次磁盘 I/O。
- 数据访问更快。聚簇索引将索引和数据保存在同一个 B-Tree 中，因此从聚簇索引中获取数据通常比在非聚簇索引中查找要快。
- 使用覆盖索引扫描的查询可以直接使用页节点中的主键值。

如果在设计表和查询时能充分利用上面的优点，那就能极大地提升性能。同时，聚簇索引也有一些缺点：

- 聚簇数据最大限度地提高了 I/O 密集型应用的性能，但如果数据全部都放在内存中，则访问的顺序就没那么重要了，聚簇索引也就没什么优势了。
- 插入速度严重依赖于插入顺序。按照主键的顺序插入是加载数据到 InnoDB 表中速度最快的方式。但如果不是按照主键顺序加载数据，那么在加载完成后最好使用 `OPTIMIZE TABLE` 命令重新组织一下表。 170
- 更新聚簇索引列的代价很高，因为会强制 InnoDB 将每个被更新的行移动到新的位置。
- 基于聚簇索引的表在插入新行，或者主键被更新导致需要移动行的时候，可能面临“页分裂（page split）”的问题。当行的主键值要求必须将这一行插入到某个已满的页中时，存储引擎会将该页分裂成两个页面来容纳该行，这就是一次页分裂操作。页分裂会导致表占用更多的磁盘空间。
- 聚簇索引可能导致全表扫描变慢，尤其是行比较稀疏，或者由于页分裂导致数据存储不连续的时候。
- 二级索引（非聚簇索引）可能比想象的要更大，因为在二级索引的叶子节点包含了引用行的主键列。

- 二级索引访问需要两次索引查找，而不是一次。

最后一点可能让人有些疑惑，为什么二级索引需要两次索引查找？答案在于二级索引中保存的“行指针”的实质。要记住，二级索引叶子节点保存的不是指向行的物理位置的指针，而是行的主键值。

这意味着通过二级索引查找行，存储引擎需要找到二级索引的叶子节点获得对应的主键值，然后根据这个值去聚簇索引中查找到对应的行。这里做了重复的工作：两次 B-Tree 查找而不是一次[注9]。对于 InnoDB，自适应哈希索引能够减少这样的重复工作。

InnoDB 和 MyISAM 的数据分布对比

聚簇索引和非聚簇索引的数据分布有区别，以及对应的主键索引和二级索引的数据分布也有区别，通常会让人感到困扰和意外。来看看 InnoDB 和 MyISAM 是如何存储下面这个表的：

```
CREATE TABLE layout_test (
   col1 int NOT NULL,
   col2 int NOT NULL,
   PRIMARY KEY(col1),
   KEY(col2)
);
```

假设该表的主键取值为 1 ~ 10 000，按照随机顺序插入并使用 `OPTIMIZE TABLE` 命令做了优化。换句话说，数据在磁盘上的存储方式已经最优，但行的顺序是随机的。列 `col2` 的值是从 1 ~ 100 之间随机赋值，所以有很多重复的值。

171 **MyISAM 的数据分布**。MyISAM 的数据分布非常简单，所以先介绍它。MyISAM 按照数据插入的顺序存储在磁盘上，如图 5-4 所示。

在行的旁边显示了行号，从 0 开始递增。因为行是定长的，所以 MyISAM 可以从表的开头跳过所需的字节找到需要的行（MyISAM 并不总是使用图 5-4 中的“行号”，而是根据定长还是变长的行使用不同策略）。

这种分布方式很容易创建索引。下面显示的一系列图，隐藏了页的物理细节，只显示索引中的“节点”，索引中的每个叶子节点包含“行号”。图 5-5 显示了表的主键。

这里忽略了一些细节，例如前一个 B-Tree 节点有多少个内部节点，不过这并不影响对非聚簇存储引擎的基本数据分布的理解。

注 9：顺便提一下，并不是所有的非聚簇索引都能做到一次索引查询就找到行。当行更新的时候可能无法存储在原来的位置，这会导致表中出现行的碎片化或者移动行并在原位置保存“向前指针”，这两种情况都会导致在查找行时需要更多的工作。

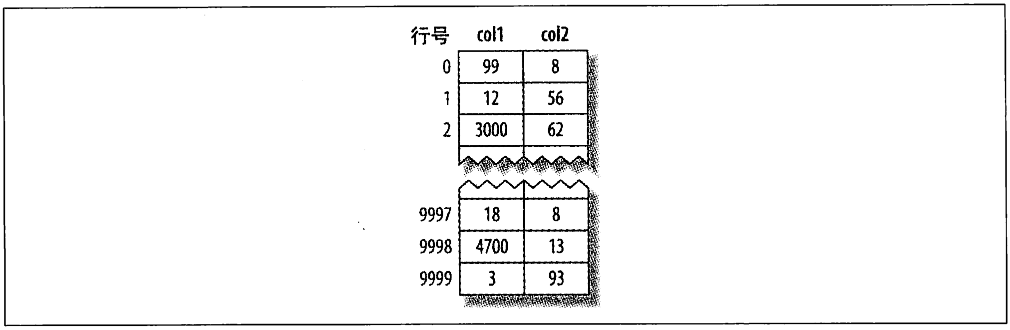


图5-4：MyISAM表layout_test的数据分布

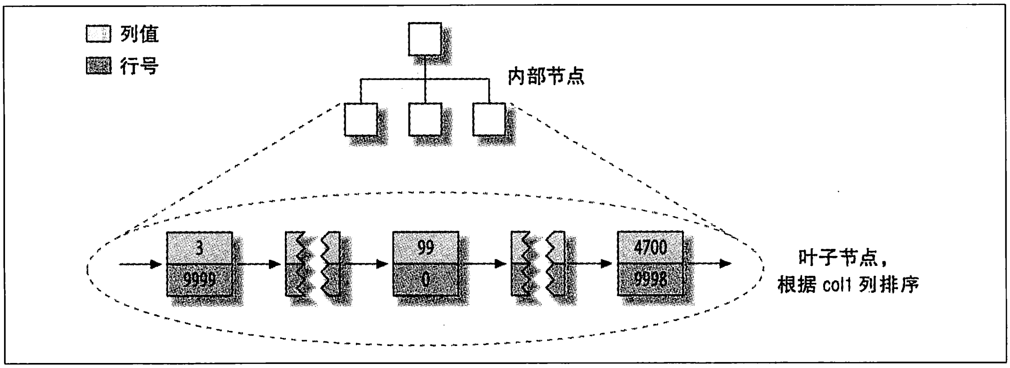


图5-5：MyISAM表layout_test的主键分布

那 col2 列上的索引又会如何呢？有什么特殊的吗？回答是否定的：它和其他索引没有什么区别。图 5-6 显示了 col2 列上的索引。

172

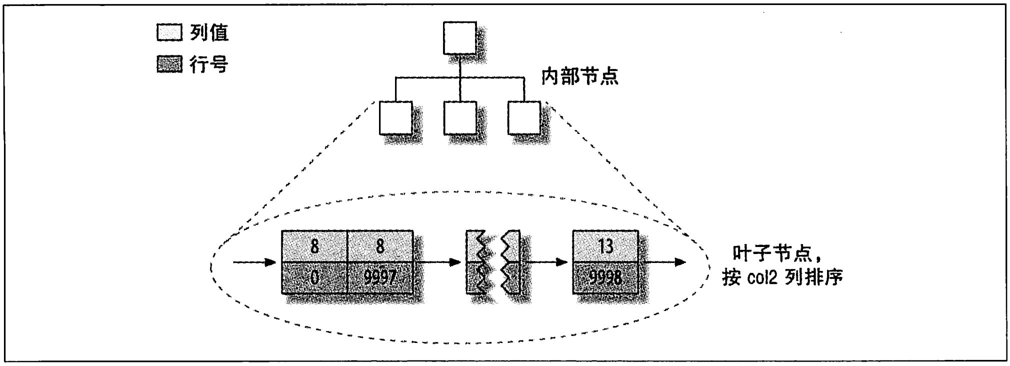


图5-6：MyISAM表layout_test的col2列索引的分布

事实上，MyISAM 中主键索引和其他索引在结构上没有什么不同。主键索引就是一个名

为 PRIMARY 的唯一非空索引。

InnoDB 的数据分布。 因为 InnoDB 支持聚簇索引，所以使用非常不同的方式存储同样的数据。InnoDB 以如图 5-7 所示的方式存储数据。

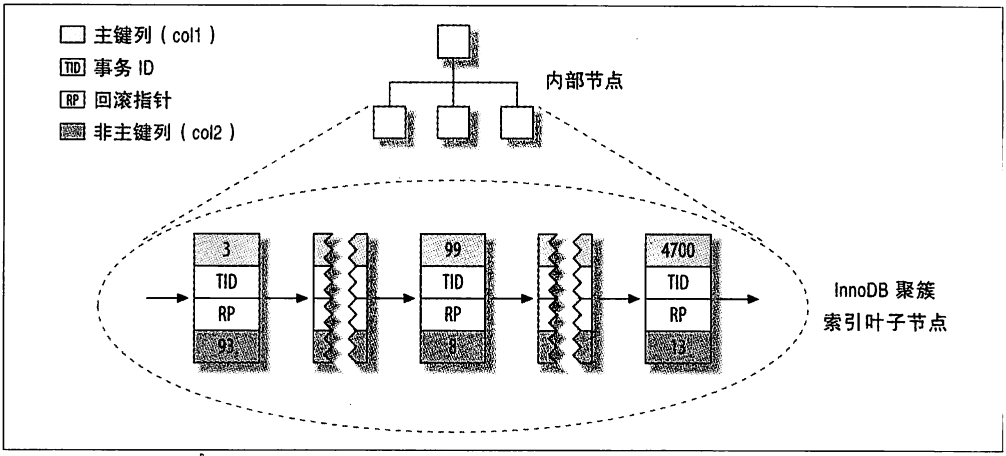


图5-7：InnoDB表layout_test的主键分布

第一眼看上去，感觉该图和前面的图 5-5 没有什么不同，但再仔细看细节，会注意到该图显示了整个表，而不是只有索引。因为在 InnoDB 中，聚簇索引"就是"表，所以不像 MyISAM 那样需要独立的行存储。

173 聚簇索引的每一个叶子节点都包含了主键值、事务 ID、用于事务和 MVCC[注10] 的回滚指针以及所有的剩余列（在这个例子中是 col2）。如果主键是一个列前缀索引，InnoDB 也会包含完整的主键列和剩下的其他列。

还有一点和 MyISAM 的不同是，InnoDB 的二级索引和聚簇索引很不相同。InnoDB 二级索引的叶子节点中存储的不是"行指针"，而是主键值，并以此作为指向行的"指针"。这样的策略减少了当出现行移动或者数据页分裂时二级索引的维护工作。使用主键值当作指针会让二级索引占用更多的空间，换来的好处是，InnoDB 在移动行时无须更新二级索引中的这个"指针"。

图 5-8 显示了示例表的 col2 索引。每一个叶子节点都包含了索引列（这里是 col2），紧接着是主键值（col1）。

图 5-8 展示了 B-Tree 的叶子节点结构，但我们故意省略了非叶子节点这样的细节。InnoDB 的非叶子节点包含了索引列和一个指向下级节点的指针（下一级节点可以是非

注 10： 多版本控制。——译者注

叶子节点，也可以是叶子节点)。这对聚簇索引和二级索引都适用。

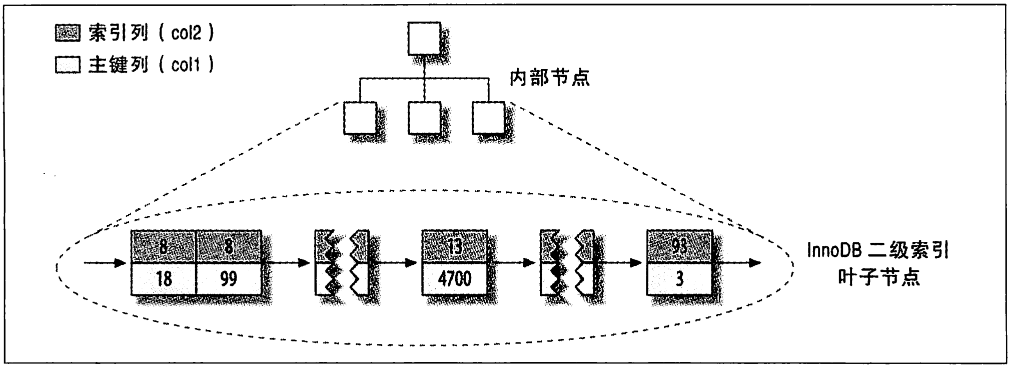


图5-8：InnoDB表layout_test的二级索引分布

图 5-9 是描述 InnoDB 和 MyISAM 如何存放表的抽象图。从图 5-9 中可以很容易看出 InnoDB 和 MyISAM 保存数据和索引的区别。

174

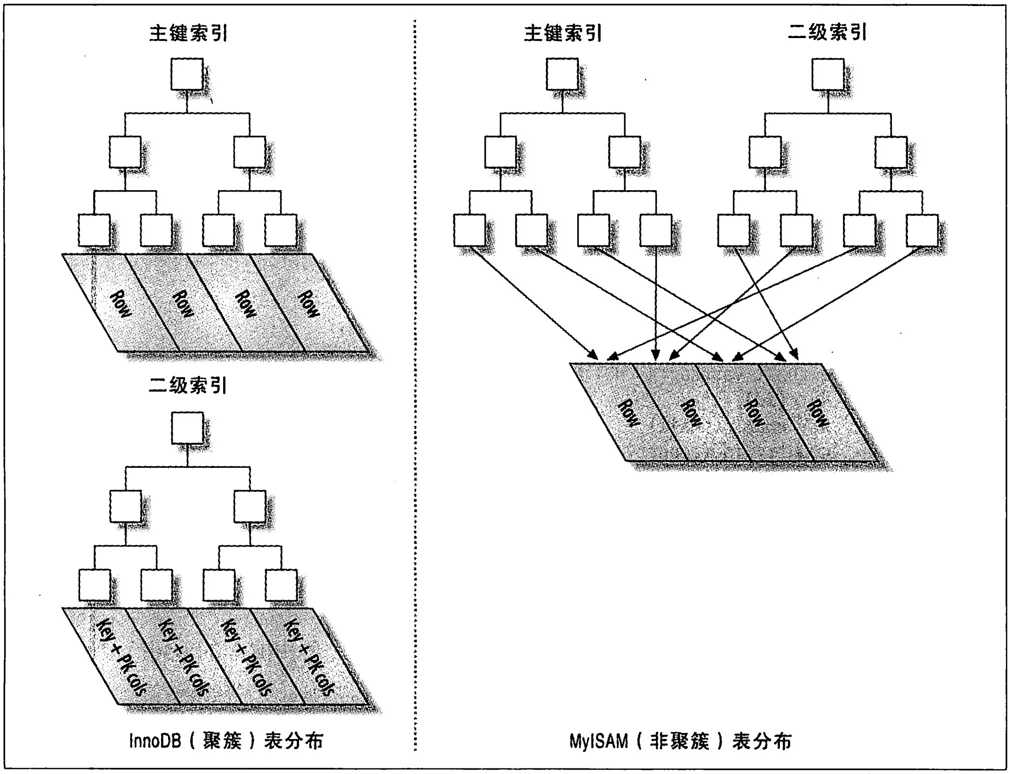


图5-9：聚簇和非聚簇表对比图

如果还没有理解聚簇索引和非聚簇索引有什么区别、为何有这些区别及这些区别的重要性，也不用担心。随着学习的深入，尤其是学完本章剩下的部分以及下一章以后，这些问题就会变得越发清楚。这些概念有些复杂，需要一些时间才能完全理解。

在 InnoDB 表中按主键顺序插入行

如果正在使用 InnoDB 表并且没有什么数据需要聚集，那么可以定义一个代理键（surrogate key）作为主键，这种主键的数据应该和应用无关，最简单的方法是使用 AUTO_INCREMENT 自增列。这样可以保证数据行是按顺序写入，对于根据主键做关联操作的性能也会更好。

最好避免随机的（不连续且值的分布范围非常大）聚簇索引，特别是对于 I/O 密集型的应用。例如，从性能的角度考虑，使用 UUID 来作为聚簇索引则会很糟糕：它使得聚簇索引的插入变得完全随机，这是最坏的情况，使得数据没有任何聚集特性。

为了演示这一点，我们做如下两个基准测试。第一个使用整数 ID 插入 userinfo 表：

```
CREATE TABLE userinfo (
   id              int unsigned NOT NULL AUTO_INCREMENT,
   name            varchar(64) NOT NULL DEFAULT '',
   email           varchar(64) NOT NULL DEFAULT '',
   password        varchar(64) NOT NULL DEFAULT '',
   dob             date DEFAULT NULL,
   address         varchar(255) NOT NULL DEFAULT '',
   city            varchar(64) NOT NULL DEFAULT '',
   state_id        tinyint unsigned NOT NULL DEFAULT '0',
   zip             varchar(8) NOT NULL DEFAULT '',
   country_id      smallint unsigned NOT NULL DEFAULT '0',
   gender          ('M','F')NOT NULL DEFAULT 'M',
   account_type    varchar(32) NOT NULL DEFAULT '',
   verified        tinyint NOT NULL DEFAULT '0',
   allow_mail      tinyint unsigned NOT NULL DEFAULT '0',
   parrent_account int unsigned NOT NULL DEFAULT '0',
   closest_airport varchar(3) NOT NULL DEFAULT '',
   PRIMARY KEY (id),
   UNIQUE  KEY email (email),
   KEY     country_id (country_id),
   KEY     state_id (state_id),
   KEY     state_id_2 (state_id,city,address)
) ENGINE=InnoDB
```

注意到使用了自增的整数 ID 作为主键[注11]。

175 第二个例子是 userinfo_uuid 表。除了主键改为 UUID，其余和前面的 userinfo 表完全相同。

注 11：值得指出的是，这是一个真实案例中的表，有很多二级索引和列。如果删除这些二级索引只测试主键，那么性能差异将会更明显。

```
CREATE TABLE userinfo_uuid (
   uuid varchar(36) NOT NULL,
   ...
```

我们测试了这两个表的设计。首先，我们在一个有足够内存容纳索引的服务器上向这两个表各插入 100 万条记录。然后向这两个表继续插入 300 万条记录，使索引的大小超过服务器的内存容量。表 5-1 对测试结果做了比较。

表 5-1：向InnoDB表插入数据的测试结果

表名	行数	时间（秒）	索引大小（MB）
userinfo	1 000 000	137	342
userinfo_uuid	1 000 000	180	544
userinfo	3 000 000	1233	1036
userinfo_uuid	3 000 000	4525	1707

注意到向 UUID 主键插入行不仅花费的时间更长，而且索引占用的空间也更大。这一方面是由于主键字段更长；另一方面毫无疑问是由于页分裂和碎片导致的。

为了明白为什么会这样，来看看往第一个表中插入数据时，索引发生了什么变化。图 5-10 显示了插满一个页面后继续插入相邻的下一个页面的场景。

176

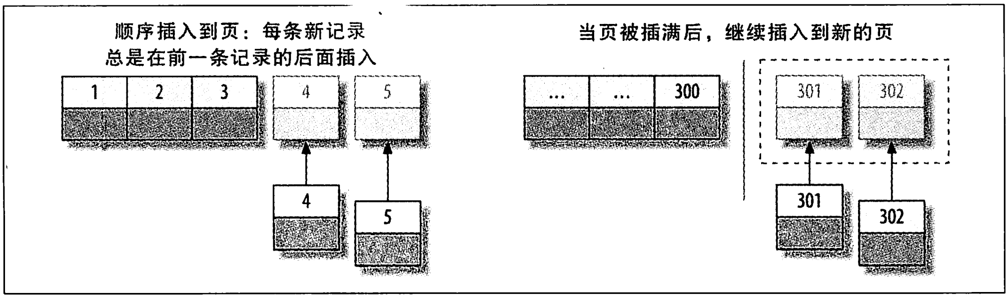


图5-10：向聚簇索引插入顺序的索引值

如图 5-10 所示，因为主键的值是顺序的，所以 InnoDB 把每一条记录都存储在上一条记录的后面。当达到页的最大填充因子时（InnoDB 默认的最大填充因子是页大小的 15/16，留出部分空间用于以后修改），下一条记录就会写入新的页中。一旦数据按照这种顺序的方式加载，主键页就会近似于被顺序的记录填满，这也正是所期望的结果（然而，二级索引页可能是不一样的）。

对比一下向第二个使用了 UUID 聚簇索引的表插入数据，看看有什么不同，图 5-11 显示了结果。

因为新行的主键值不一定比之前插入的大，所以 InnoDB 无法简单地总是把新行插入到索引的最后，而是需要为新的行寻找合适的位置——通常是已有数据的中间位置——并且分配空间。这会增加很多的额外工作，并导致数据分布不够优化。下面是总结的一些缺点：

- 写入的目标页可能已经刷到磁盘上并从缓存中移除，或者是还没有被加载到缓存中，InnoDB 在插入之前不得不先找到并从磁盘读取目标页到内存中。这将导致大量的随机 I/O。
- 因为写入是乱序的，InnoDB 不得不频繁地做页分裂操作，以便为新的行分配空间。页分裂会导致移动大量数据，一次插入最少需要修改三个页而不是一个页。
- 由于频繁的页分裂，页会变得稀疏并被不规则地填充，所以最终数据会有碎片。

在把这些随机值载入到聚簇索引以后，也许需要做一次 OPTIMIZE TABLE 来重建表并优化页的填充。

从这个案例可以看出，使用 InnoDB 时应该尽可能地按主键顺序插入数据，并且尽可能地使用单调增加的聚簇键的值来插入新行。

177

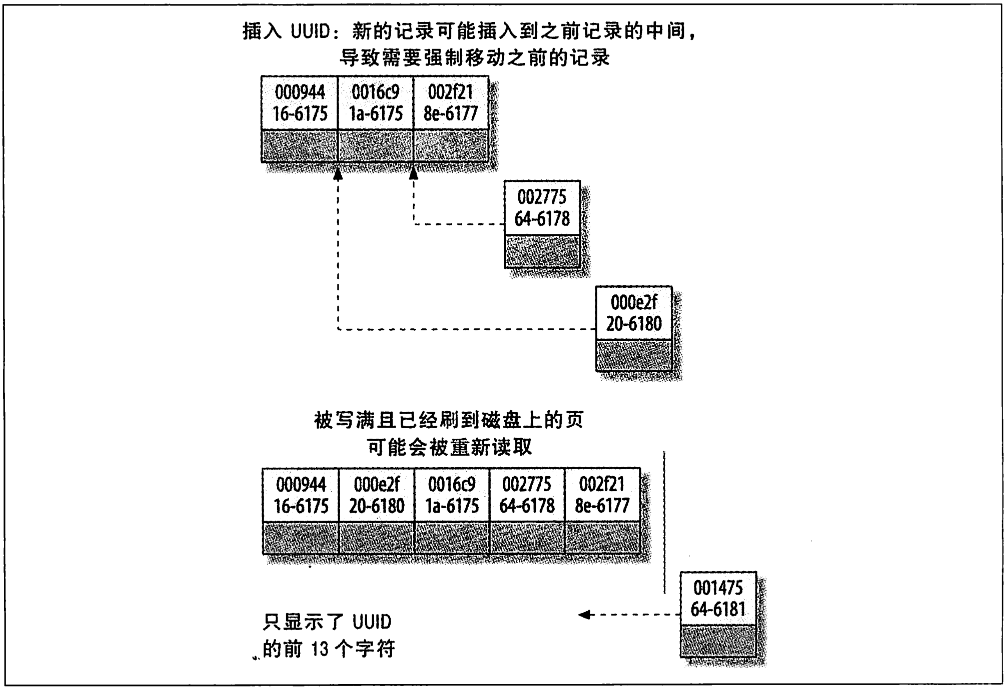


图5-11：向聚簇索引中插入无序的值

顺序的主键什么时候会造成更坏的结果？

对于高并发工作负载，在 InnoDB 中按主键顺序插入可能会造成明显的争用。主键的上界会成为“热点”。因为所有的插入都发生在这里，所以并发插入可能导致间隙锁竞争。另一个热点可能是 AUTO_INCREMENT 锁机制；如果遇到这个问题，则可能需要考虑重新设计表或者应用，或者更改 innodb_autoinc_lock_mode 配置。如果你的服务器版本还不支持 innodb_autoinc_lock_mode 参数，可以升级到新版本的 InnoDB，可能对这种场景会工作得更好。

5.3.6 覆盖索引

通常大家都会根据查询的 WHERE 条件来创建合适的索引，不过这只是索引优化的一个方面。设计优秀的索引应该考虑到整个查询，而不单单是 WHERE 条件部分。索引确实是一种查找数据的高效方式，但是 MySQL 也可以使用索引来直接获取列的数据，这样就不再需要读取数据行。如果索引的叶子节点中已经包含要查询的数据，那么还有什么必要再回表查询呢？如果一个索引包含（或者说覆盖）所有需要查询的字段的值，我们就称 178
之为“覆盖索引”。

覆盖索引是非常有用的工具，能够极大地提高性能。考虑一下如果查询只需要扫描索引而无须回表，会带来多少好处：

- 索引条目通常远小于数据行大小，所以如果只需要读取索引，那 MySQL 就会极大地减少数据访问量。这对缓存的负载非常重要，因为这种情况下响应时间大部分花费在数据拷贝上。覆盖索引对于 I/O 密集型的应用也有帮助，因为索引比数据更小，更容易全部放入内存中（这对于 MyISAM 尤其正确，因为 MyISAM 能压缩索引以变得更小）。
- 因为索引是按照列值顺序存储的（至少在单个页内是如此），所以对于 I/O 密集型的范围查询会比随机从磁盘读取每一行数据的 I/O 要少得多。对于某些存储引擎，例如 MyISAM 和 Percona XtraDB，甚至可以通过 OPTIMIZE 命令使得索引完全顺序排列，这让简单的范围查询能使用完全顺序的索引访问。
- 一些存储引擎如 MyISAM 在内存中只缓存索引，数据则依赖于操作系统来缓存，因此要访问数据需要一次系统调用。这可能会导致严重的性能问题，尤其是那些系统调用占了数据访问中的最大开销的场景。
- 由于 InnoDB 的聚簇索引，覆盖索引对 InnoDB 表特别有用。InnoDB 的二级索引在叶子节点中保存了行的主键值，所以如果二级主键能够覆盖查询，则可以避免对主键索引的二次查询。

在所有这些场景中，在索引中满足查询的成本一般比查询行要小得多。

不是所有类型的索引都可以成为覆盖索引。覆盖索引必须要存储索引列的值，而哈希索引、空间索引和全文索引等都不存储索引列的值，所以 MySQL 只能使用 B-Tree 索引做覆盖索引。另外，不同的存储引擎实现覆盖索引的方式也不同，而且不是所有的引擎都支持覆盖索引（在写作本书时，Memory 存储引擎就不支持覆盖索引）。

当发起一个被索引覆盖的查询（也叫做索引覆盖查询）时，在 EXPLAIN 的 Extra 列可以看到“Using index”的信息[注12]。例如，表 sakila.inventory 有一个多列索引 (store_id, film_id)。MySQL 如果只需访问这两列，就可以使用这个索引做覆盖索引，如下所示：

```
mysql> EXPLAIN SELECT store_id, film_id FROM sakila.inventory\G
*************************** 1. row ***************************
           id: 1
179  select_type: SIMPLE
        table: inventory
         type: index
possible_keys: NULL
          key: idx_store_id_film_id
      key_len: 3
          ref: NULL
         rows: 4673
        Extra: Using index
```

索引覆盖查询还有很多陷阱可能会导致无法实现优化。MySQL 查询优化器会在执行查询前判断是否有一个索引能进行覆盖。假设索引覆盖了 WHERE 条件中的字段，但不是整个查询涉及的字段。如果条件为假（false），MySQL 5.5 和更早的版本也总是会回表获取数据行，尽管并不需要这一行且最终会被过滤掉。

来看看为什么会发生这样的情况，以及如何重写查询以解决该问题。从下面的查询开始：

```
mysql> EXPLAIN SELECT * FROM products WHERE actor='SEAN CARREY'
    -> AND title like '%APOLLO%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: products
         type: ref
possible_keys: ACTOR,IX_PROD_ACTOR
          key: ACTOR
      key_len: 52
          ref: const
         rows: 10
        Extra: Using where
```

注 12： 很容易把 Extra 列的“Using index”和 type 列的“index”搞混淆。其实这两者完全不同，type 列和覆盖索引毫无关系；它只是表示这个查询访问数据的方式，或者说是 MySQL 查找行的方式。MySQL 手册中称之为连接方式（join type）。

这里索引无法覆盖该查询，有两个原因：

- 没有任何索引能够覆盖这个查询。因为查询从表中选择了所有的列，而没有任何索引覆盖了所有的列。不过，理论上 MySQL 还有一个捷径可以利用：WHERE 条件中的列是有索引可以覆盖的，因此 MySQL 可以使用该索引找到对应的 actor 并检查 title 是否匹配，过滤之后再读取需要的数据行。
- MySQL 不能在索引中执行 LIKE 操作。这是底层存储引擎 API 的限制，MySQL 5.5 和更早的版本中只允许在索引中做简单比较操作（例如等于、不等于以及大于）。MySQL 能在索引中做最左前缀匹配的 LIKE 比较，因为该操作可以转换为简单的比较操作，但是如果是通配符开头的 LIKE 查询，存储引擎就无法做比较匹配。这种情况下，MySQL 服务器只能提取数据行的值而不是索引值来做比较。

也有办法可以解决上面说的两个问题，需要重写查询并巧妙地设计索引。先将索引扩展至覆盖三个数据列（artist, title, prod_id），然后按如下方式重写查询：

180

```
mysql> EXPLAIN SELECT *
    -> FROM products
    ->    JOIN (
    ->       SELECT prod_id
    ->       FROM products
    ->       WHERE actor='SEAN CARREY' AND title LIKE '%APOLLO%'
    ->    ) AS t1 ON (t1.prod_id=products.prod_id)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
               ...omitted...
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: products
               ...omitted...
*************************** 3. row ***************************
           id: 2
  select_type: DERIVED
        table: products
         type: ref
possible_keys: ACTOR,ACTOR_2,IX_PROD_ACTOR
          key: ACTOR_2
      key_len: 52
          ref:
         rows: 11
        Extra: Using where; Using index
```

我们把这种方式叫做延迟关联（deferred join），因为延迟了对列的访问。在查询的第一阶段 MySQL 可以使用覆盖索引，在 FROM 子句的子查询中找到匹配的 prod_id，然后根

据这些 prod_id 值在外层查询匹配获取需要的所有列值。虽然无法使用索引覆盖整个查询，但总算比完全无法利用索引覆盖的好。

这样优化的效果取决于 WHERE 条件匹配返回的行数。假设这个 products 表有 100 万行，我们来看一下上面两个查询在三个不同的数据集上的表现，每个数据集都包含 100 万行：

1. 第一个数据集，Sean Carrey 出演了 30 000 部作品，其中有 20 000 部的标题中包含了 Apollo。
2. 第二个数据集，Sean Carrey 出演了 30 000 部作品，其中 40 部的标题中包含了 Apollo。
3. 第三个数据集，Sean Carrey 出演了 50 部作品，其中 10 部的标题中包含了 Apollo。

使用上面的三种数据集来测试两种不同的查询，得到的结果如表 5-2 所示。

181 表5-2：索引覆盖查询和非覆盖查询的测试结果

数据集	原查询	优化后的查询
示例 1	每秒 5 次查询	每秒 5 次查询
示例 2	每秒 7 次查询	每秒 35 次查询
示例 3	每秒 2 400 次查询	每秒 2 000 次查询

下面是对结果的分析 ：

- 在示例 1 中，查询返回了一个很大的结果集，因此看不到优化的效果。大部分时间都花在读取和发送数据上了。
- 在示例 2 中，经过索引过滤，尤其是第二个条件过滤后只返回了很少的结果集，优化的效果非常明显 ：在这个数据集上性能提高了 5 倍，优化后的查询的效率主要得益于只需要读取 40 行完整数据行，而不是原查询中需要的 30 000 行。
- 在示例 3 中，显示了子查询效率反而下降的情况。因为索引过滤时符合第一个条件的结果集已经很小，所以子查询带来的成本反而比从表中直接提取完整行更高。

在大多数存储引擎中，覆盖索引只能覆盖那些只访问索引中部分列的查询。不过，可以更进一步优化 InnoDB。回想一下，InnoDB 的二级索引的叶子节点都包含了主键的值，这意味着 InnoDB 的二级索引可以有效地利用这些“额外”的主键列来覆盖查询。

例如，sakila.actor 使用 InnoDB 存储引擎，并在 last_name 字段有二级索引，虽然该索引的列不包括主键 actor_id，但也能够用于对 actor_id 做覆盖查询 ：

```
mysql> EXPLAIN SELECT actor_id, last_name
    -> FROM sakila.actor WHERE last_name = 'HOPPER'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: ref
possible_keys: idx_actor_last_name
          key: idx_actor_last_name
      key_len: 137
          ref: const
         rows: 2
        Extra: Using where; Using index
```

未来 MySQL 版本的改进

182

上面提到的很多限制都是由于存储引擎 API 设计所导致的，目前的 API 设计不允许 MySQL 将过滤条件传到存储引擎层。如果 MySQL 在后续版本能够做到这一点，则可以把查询发送到数据上，而不是像现在这样只能把数据从存储引擎拉到服务器层，再根据查询条件过滤。在本书写作之际，MySQL 5.6 版本（未正式发布）包含了在存储引擎 API 上所做的一个重要的改进，其被称为“索引条件推送（index condition pushdown）”。这个特性将大大改善现在的查询执行方式，如此一来上面介绍的很多技巧也就不再需要了。

5.3.7 使用索引扫描来做排序

MySQL 有两种方式可以生成有序的结果：通过排序操作；或者按索引顺序扫描[注13]；如果 EXPLAIN 出来的 type 列的值为“index”，则说明 MySQL 使用了索引扫描来做排序（不要和 Extra 列的“Using index”搞混淆了）。

扫描索引本身是很快的，因为只需要从一条索引记录移动到紧接着的下一条记录。但如果索引不能覆盖查询所需的全部列，那就不得不每扫描一条索引记录就都回表查询一次对应的行。这基本上都是随机 I/O，因此按索引顺序读取数据的速度通常要比顺序地全表扫描慢，尤其是在 I/O 密集型的工作负载时。

MySQL 可以使用同一个索引既满足排序，又用于查找行。因此，如果可能，设计索引时应该尽可能地同时满足这两种任务，这样是最好的。

只有当索引的列顺序和 ORDER BY 子句的顺序完全一致，并且所有列的排序方向（倒序

注 13： MySQL 有两种排序算法，更多细节可以阅读第 7 章。

或正序）都一样时，MySQL 才能够使用索引来对结果做排序[注14]。如果查询需要关联多张表，则只有当 ORDER BY 子句引用的字段全部为第一个表时，才能使用索引做排序。ORDER BY 子句和查找型查询的限制是一样的：需要满足索引的最左前缀的要求；否则，MySQL 都需要执行排序操作，而无法利用索引排序。

有一种情况下 ORDER BY 子句可以不满足索引的最左前缀的要求，就是前导列为常量的时候。如果 WHERE 子句或者 JOIN 子句中对这些列指定了常量，就可以“弥补”索引的不足。

例如，Sakila 示例数据库的表 rental 在列（rental_date, inventory_id, customer_id）上有名为 rental_date 的索引。

(rental_date, inventory_id, customer_id):

183

```
CREATE TABLE rental (
   ...
   PRIMARY KEY (rental_id),
   UNIQUE KEY rental_date (rental_date,inventory_id,customer_id),
   KEY idx_fk_inventory_id (inventory_id),
   KEY idx_fk_customer_id (customer_id),
   KEY idx_fk_staff_id (staff_id),
   ...
);
```

MySQL 可以使用 rental_date 索引为下面的查询做排序，从 EXPLAIN 中可以看到没有出现文件排序（filesort）操作[注15]：

```
mysql> EXPLAIN SELECT rental_id, staff_id FROM sakila.rental
    -> WHERE rental_date = '2005-05-25'
    -> ORDER BY inventory_id, customer_id\G
*************************** 1. row ***************************
         type: ref
possible_keys: rental_date
          key: rental_date
         rows: 1
        Extra: Using where
```

即使 ORDER BY 子句不满足索引的最左前缀的要求，也可以用于查询排序，这是因为索引的第一列被指定为一个常数。

还有更多可以使用索引做排序的查询示例。下面这个查询可以利用索引排序，是因为查询为索引的第一列提供了常量条件，而使用第二列进行排序，将两列组合在一起，就形成了索引的最左前缀：

```
... WHERE rental_date = '2005-05-25' ORDER BY inventory_id DESC;
```

注 14： 如果需要按不同方向做排序，一个技巧是存储该列值的反转串或者相反数。
注 15： MySQL 这里称其为文件排序（filesort），其实并不一定使用磁盘文件。

下面这个查询也没问题，因为 ORDER BY 使用的两列就是索引的最左前缀：

```
... WHERE rental_date > '2005-05-25' ORDER BY rental_date, inventory_id;
```

下面是一些不能使用索引做排序的查询：

- 下面这个查询使用了两种不同的排序方向，但是索引列都是正序排序的：

  ```
  ... WHERE rental_date = '2005-05-25' ORDER BY inventory_id DESC, customer_id ASC;
  ```

- 下面这个查询的 ORDER BY 子句中引用了一个不在索引中的列：

  ```
  ... WHERE rental_date = '2005-05-25' ORDER BY inventory_id, staff_id;
  ```

- 下面这个查询的 WHERE 和 ORDER BY 中的列无法组合成索引的最左前缀：

  ```
  ... WHERE rental_date = '2005-05-25' ORDER BY customer_id;
  ```

- 下面这个查询在索引列的第一列上是范围条件，所以 MySQL 无法使用索引的其余列：

  ```
  ... WHERE rental_date > '2005-05-25' ORDER BY inventory_id, customer_id;
  ```

- 这个查询在 inventory_id 列上有多个等于条件。对于排序来说，这也是一种范围查询：

  ```
  ... WHERE rental_date = '2005-05-25' AND inventory_id IN(1,2) ORDER BY customer_
  id;
  ```

184

下面这个例子理论上是可以使用索引进行关联排序的，但由于优化器在优化时将 film_actor 表当作关联的第二张表，所以实际上无法使用索引：

```
mysql> EXPLAIN SELECT actor_id, title FROM sakila.film_actor
    -> INNER JOIN sakila.film USING(film_id) ORDER BY actor_id\G
+------------+------------------------------------------------+
| table      | Extra                                          |
+------------+------------------------------------------------+
| film       | Using index; Using temporary; Using filesort |
| film_actor | Using index                                    |
+------------+------------------------------------------------+
```

使用索引做排序的一个最重要的用法是当查询同时有 ORDER BY 和 LIMIT 子句的时候。后面我们会具体介绍这些内容。

5.3.8 压缩（前缀压缩）索引

MyISAM 使用前缀压缩来减少索引的大小，从而让更多的索引可以放入内存中，这在某些情况下能极大地提高性能。默认只压缩字符串，但通过参数设置也可以对整数做压缩。

MyISAM 压缩每个索引块的方法是，先完全保存索引块中的第一个值，然后将其他值和第一个值进行比较得到相同前缀的字节数和剩余的不同后缀部分，把这部分存储起来即可。例如，索引块中的第一个值是“perform”，第二个值是“performance”，那么第二个值的前缀压缩后存储的是类似“7,ance”这样的形式。MyISAM 对行指针也采用类似的前缀压缩方式。

压缩块使用更少的空间，代价是某些操作可能更慢。因为每个值的压缩前缀都依赖前面的值，所以 MyISAM 查找时无法在索引块使用二分查找而只能从头开始扫描。正序的扫描速度还不错，但是如果是倒序扫描——例如 ORDER BY DESC——就不是很好了。所有在块中查找某一行的操作平均都需要扫描半个索引块。

测试表明，对于 CPU 密集型应用，因为扫描需要随机查找，压缩索引使得 MyISAM 在索引查找上要慢好几倍。压缩索引的倒序扫描就更慢了。压缩索引需要在 CPU 内存资源与磁盘之间做权衡。压缩索引可能只需要十分之一大小的磁盘空间，如果是 I/O 密集型应用，对某些查询带来的好处会比成本多很多。

可以在 CREATE TABLE 语句中指定 PACK_KEYS 参数来控制索引压缩的方式。

185

5.3.9 冗余和重复索引

MySQL 允许在相同列上创建多个索引，无论是有意的还是无意的。MySQL 需要单独维护重复的索引，并且优化器在优化查询的时候也需要逐个地进行考虑，这会影响性能。

重复索引是指在相同的列上按照相同的顺序创建的相同类型的索引。应该避免这样创建重复索引，发现以后也应该立即移除。

有时会在不经意间创建了重复索引，例如下面的代码：

```
CREATE TABLE test (
   ID INT NOT NULL PRIMARY KEY,
   A  INT NOT NULL,
   B  INT NOT NULL,
   UNIQUE(ID),
   INDEX(ID)
) ENGINE=InnoDB;
```

一个经验不足的用户可能是想创建一个主键，先加上唯一限制，然后再加上索引以供查询使用。事实上，MySQL 的唯一限制和主键限制都是通过索引实现的，因此，上面的写法实际上在相同的列上创建了三个重复的索引。通常并没有理由这样做，除非是在同一列上创建不同类型的索引来满足不同的查询需求[注16]。

注 16： 如果索引类型不同，并不算是重复索引。例如经常有很好的理由创建 KEY(*col*) 和 FULLTEXT KEY(*col*) 两种索引。

冗余索引和重复索引有一些不同。如果创建了索引 (A, B)，再创建索引 (A) 就是冗余索引，因为这只是前一个索引的前缀索引。因此索引 (A, B) 也可以当作索引 (A) 来使用（这种冗余只是对 B-Tree 索引来说的）。但是如果再创建索引 (B, A)，则不是冗余索引，索引 (B) 也不是，因为 B 不是索引 (A, B) 的最左前缀列。另外，其他不同类型的索引（例如哈希索引或者全文索引）也不会是 B-Tree 索引的冗余索引，而无论覆盖的索引列是什么。

冗余索引通常发生在为表添加新索引的时候。例如，有人可能会增加一个新的索引 (A, B) 而不是扩展已有的索引 (A)。还有一种情况是将一个索引扩展为 (A, ID)，其中 ID 是主键，对于 InnoDB 来说主键列已经包含在二级索引中了，所以这也是冗余的。

大多数情况下都不需要冗余索引，应该尽量扩展已有的索引而不是创建新索引。但也有时候出于性能方面的考虑需要冗余索引，因为扩展已有的索引会导致其变得太大，从而影响其他使用该索引的查询的性能。

例如，如果在整数列上有一个索引，现在需要额外增加一个很长的 VARCHAR 列来扩展该索引，那性能可能会急剧下降。特别是有查询把这个索引当作覆盖索引，或者这是 MyISAM 表并且有很多范围查询（由于 MyISAM 的前缀压缩）的时候。 186

考虑一下前面“在 InnoDB 中按主键顺序插入行”一节提到的 userinfo 表。这个表有 1 000 000 行，对每个 state_id 值大概有 20 000 条记录。在 state_id 列有一个索引对下面的查询有用，假设查询名为 Q1 ：

```
mysql> SELECT count(*) FROM userinfo WHERE state_id=5;
```

一个简单的测试表明该查询的执行速度大概是每秒 115 次 (QPS)。还有一个相关查询需要检索几个列的值，而不是只统计行数，假设名为 Q2 ：

```
mysql> SELECT state_id, city, address FROM userinfo WHERE state_id=5;
```

对于这个查询，测试结果 QPS 小于 10[注17]。提升该查询性能的最简单办法就是扩展索引为 (state_id, city, address)，让索引能覆盖查询 ：

```
mysql> ALTER TABLE userinfo DROP KEY state_id,
    ->     ADD KEY state_id_2 (state_id, city, address);
```

索引扩展后，Q2 运行得更快了，但是 Q1 却变慢了。如果我们想让两个查询都变得更快，就需要两个索引，尽管这样一来原来的单列索引是冗余的了。表 5-3 显示这两个查询在不同的索引策略下的详细结果，分别使用 MyISAM 和 InnoDB 存储引擎。注意到只有

注 17 ： 这里使用了全内存的案例，如果表逐渐变大，导致工作负载变成 I/O 密集型时，性能测试结果差距会更大。对于 COUNT() 查询，覆盖索引性能提升 100 倍也是很有可能的。

state_id_2 索引时，InnoDB 引擎上的查询 Q1 的性能下降并不明显，这是因为 InnoDB 没有使用索引压缩。

表5-3：使用不同索引策略的SELECT查询的QPS测试结果

	只有state_id	只有state_id_2	同时有state_id和state_id_2
MyISAM, Q1	114.96	25.40	112.19
MyISAM, Q2	9.97	16.34	16.37
InnoDB, Q1	108.55	100.33	107.97
InnoDB, Q2	12.12	28.04	28.06

有两个索引的缺点是索引成本更高。表 5-4 显示了向表中插入 100 万行数据所需要的时间。

187

表5-4：在使用不同索引策略时插入100万行数据的速度

	只有state_id	同时有state_id和state_id_2
InnoDB，对两个索引都有足够的内容	80 秒	136 秒
MyISAM，只有一个索引有足够的内容	72 秒	470 秒

可以看到，表中的索引越多插入速度会越慢。一般来说，增加新索引将会导致 INSERT、UPDATE、DELETE 等操作的速度变慢，特别是当新增索引后导致达到了内存瓶颈的时候。

解决冗余索引和重复索引的方法很简单，删除这些索引就可以，但首先要做的是找出这样的索引。可以通过写一些复杂的访问 INFORMATION_SCHEMA 表的查询来找，不过还有两个更简单的方法。可使用 Shlomi Noach 的 *common_schema* 中的一些视图来定位，*common_schema* 是一系列可以安装到服务器上的常用的存储和视图（*http://code.google.com/p/common-schema/*）。这比自己编写查询要快而且简单。另外也可以使用 Percona Toolkit 中的 *pt-duplicate-key-checker*，该工具通过分析表结构来找出冗余和重复的索引。对于大型服务器来说，使用外部的工具可能更合适些；如果服务器上有大量的数据或者大量的表，查询 INFORMATION_SCHEMA 表可能会导致性能问题。

在决定哪些索引可以被删除的时候要非常小心。回忆一下，在前面的 InnoDB 的示例表中，因为二级索引的叶子节点包含了主键值，所以在列 (A) 上的索引就相当于在 (A, ID) 上的索引。如果有像 WHERE A = 5 ORDER BY ID 这样的查询，这个索引会很有作用。但如果将索引扩展为 (A, B)，则实际上就变成了 (A, B, ID)，那么上面查询的 ORDER BY 子句就无法使用该索引做排序，而只能用文件排序了。所以，建议使用 Percona 工具箱中的 *pt-upgrade* 工具来仔细检查计划中的索引变更。

5.3.10 未使用的索引

除了冗余索引和重复索引，可能还会有一些服务器永远不用的索引。这样的索引完全是累赘，建议考虑删除[注18]。有两个工具可以帮助定位未使用的索引。最简单有效的办法是在 Percona Server 或者 MariaDB 中先打开 userstates 服务器变量（默认是关闭的），然后让服务器正常运行一段时间，再通过查询 INFORMATION_SCHEMA.INDEX_STATISTICS 就能查到每个索引的使用频率。

另外，还可以使用 Percona Toolkit 中的 *pt-index-usage*，该工具可以读取查询日志，并对日志中的每条查询进行 EXPLAIN 操作，然后打印出关于索引和查询的报告。这个工具不仅可以找出哪些索引是未使用的，还可以了解查询的执行计划——例如在某些情况有些类似的查询的执行方式不一样，这可以帮助你定位到那些偶尔服务质量差的查询，优化它们以得到一致的性能表现。该工具也可以将结果写入到 MySQL 的表中，方便查询结果。 188

5.3.11 索引和锁

索引可以让查询锁定更少的行。如果你的查询从不访问那些不需要的行，那么就会锁定更少的行，从两个方面来看这对性能都有好处。首先，虽然 InnoDB 的行锁效率很高，内存使用也很少，但是锁定行的时候仍然会带来额外开销；其次，锁定超过需要的行会增加锁争用并减少并发性。

InnoDB 只有在访问行的时候才会对其加锁，而索引能够减少 InnoDB 访问的行数，从而减少锁的数量。但这只有当 InnoDB 在存储引擎层能够过滤掉所有不需要的行时才有效。如果索引无法过滤掉无效的行，那么在 InnoDB 检索到数据并返回给服务器层以后，MySQL 服务器才能应用 WHERE 子句[注19]。这时已经无法避免锁定行了：InnoDB 已经锁住了这些行，到适当的时候才释放。在 MySQL 5.1 和更新的版本中，InnoDB 可以在服务器端过滤掉行后就释放锁，但是在早期的 MySQL 版本中，InnoDB 只有在事务提交后才能释放锁。

通过下面的例子再次使用数据库 Sakila 很好地解释了这些情况：

```
mysql> SET AUTOCOMMIT=0;
mysql> BEGIN;
mysql> SELECT actor_id FROM sakila.actor WHERE actor_id < 5
    ->    AND actor_id <> 1 FOR UPDATE;
```

注 18： 有些索引的功能相当于唯一约束，虽然该索引一直没有被查询使用，却可能是用于避免产生重复数据的。

注 19： 再说一下，MySQL 5.6 对于这里的问题可能会有很大的帮助。

```
+----------+
| actor_id |
+----------+
|        2 |
|        3 |
|        4 |
+----------+
```

这条查询仅仅会返回 2 ~ 4 之间的行，但是实际上获取了 1 ~ 4 之间的行的排他锁。InnoDB 会锁住第 1 行，这是因为 MySQL 为该查询选择的执行计划是索引范围扫描：

189

```
mysql> EXPLAIN SELECT actor_id FROM sakila.actor
    -> WHERE actor_id < 5 AND actor_id <> 1 FOR UPDATE;
+----+-------------+-------+-------+---------+--------------------------+
| id | select_type | table | type  | key     | Extra                    |
+----+-------------+-------+-------+---------+--------------------------+
|  1 | SIMPLE      | actor | range | PRIMARY | Using where; Using index |
+----+-------------+-------+-------+---------+--------------------------+
```

换句话说，底层存储引擎的操作是“从索引的开头开始获取满足条件 `actor_id < 5` 的记录”，服务器并没有告诉 InnoDB 可以过滤第 1 行的 `WHERE` 条件。注意到 `EXPLAIN` 的 `Extra` 列出现了“Using where”，这表示 MySQL 服务器将存储引擎返回行以后再应用 `WHERE` 过滤条件。

下面的第二个查询就能证明第 1 行确实已经被锁定，尽管第一个查询的结果中并没有这个第 1 行。保持第一个连接打开，然后开启第二个连接并执行如下查询：

```
mysql> SET AUTOCOMMIT=0;
mysql> BEGIN;
mysql> SELECT actor_id FROM sakila.actor WHERE actor_id = 1 FOR UPDATE;
```

这个查询将会挂起，直到第一个事务释放第 1 行的锁。这个行为对于基于语句的复制（将在第 10 章讨论）的正常运行来说是必要的。[注 20]

就像这个例子显示的，即使使用了索引，InnoDB 也可能锁住一些不需要的数据。如果不能使用索引查找和锁定行的话问题可能会更糟糕，MySQL 会做全表扫描并锁住所有的行，而不管是不是需要。

关于 InnoDB、索引和锁有一些很少有人知道的细节：InnoDB 在二级索引上使用共享（读）锁，但访问主键索引需要排他（写）锁。这消除了使用覆盖索引的可能性，并且使得 `SELECT FOR UPDATE` 比 `LOCK IN SHARE MODE` 或非锁定查询要慢很多。

注 20： 尽管理论上使用基于行的日志模式时，在某些事务隔离级别下，服务器不再需要锁定行，但实践中经常发现无法实现这种预期的行为。直到 MySQL 5.6.3 版本，在 read-commit 隔离级别和基于行的日志模式下，这个例子还是会导致锁。

5.4 索引案例学习

理解索引最好的办法是结合示例，所以这里准备了一个索引的案例。

假设要设计一个在线约会网站，用户信息表有很多列，包括国家、地区、城市、性别、眼睛颜色，等等。网站必须支持上面这些特征的各种组合来搜索用户，还必须允许根据用户的最后在线时间、其他会员对用户的评分等对用户进行排序并对结果进行限制。如何设计索引满足上面的复杂需求呢？

出人意料的是第一件需要考虑的事情是需要使用索引来排序，还是先检索数据再排序。 190
使用索引排序会严格限制索引和查询的设计。例如，如果希望使用索引做根据其他会员对用户的评分的排序，则 WHERE 条件中的 age BETWEEN 18 AND 25 就无法使用索引。如果 MySQL 使用某个索引进行范围查询，也就无法再使用另一个索引（或者是该索引的后续字段）进行排序了。如果这是很常见的 WHERE 条件，那么我们当然就会认为很多查询需要做排序操作（例如文件排序 filesort）。

5.4.1 支持多种过滤条件

现在需要看看哪些列拥有很多不同的取值，哪些列在 WHERE 子句中出现得最频繁。在有更多不同值的列上创建索引的选择性会更好。一般来说这样做都是对的，因为可以让 MySQL 更有效地过滤掉不需要的行。

country 列的选择性通常不高，但可能很多查询都会用到。sex 列的选择性肯定很低，但也会在很多查询中用到。所以考虑到使用的频率，还是建议在创建不同组合索引的时候将 (sex, country) 列作为前缀。

但根据传统的经验不是说不应该在选择性低的列上创建索引的吗？那为什么这里要将两个选择性都很低的字段作为索引的前缀列？我们的脑子坏了？

我们的脑子当然没坏。这么做有两个理由：第一点，如前所述几乎所有的查询都会用到 sex 列。前面曾提到，几乎每一个查询都会用到 sex 列，甚至会把网站设计成每次都只能按某一种性别搜索用户。更重要的一点是，索引中加上这一列也没有坏处，即使查询没有使用 sex 列也可以通过下面的“诀窍”绕过。

这个“诀窍”就是：如果某个查询不限制性别，那么可以通过在查询条件中新增 AND SEX IN('m','f') 来让 MySQL 选择该索引。这样写并不会过滤任何行，和没有这个条件时返回的结果相同。但是必须加上这个列的条件，MySQL 才能够匹配索引的最左前缀。这个“诀窍”在这类场景中非常有效，但如果列有太多不同的值，就会让 IN() 列表太长，这样做就不行了。

这个案例显示了一个基本原则：考虑表上所有的选项。当设计索引时，不要只为现有的查询考虑需要哪些索引，还需要考虑对查询进行优化。如果发现某些查询需要创建新索引，但是这个索引又会降低另一些查询的效率，那么应该想一下是否能优化原来的查询。应该同时优化查询和索引以找到最佳的平衡，而不是闭门造车去设计最完美的索引。

191 接下来，需要考虑其他常见 WHERE 条件的组合，并需要了解哪些组合在没有合适索引的情况下会很慢。(sex, country, age) 上的索引就是一个很明显的选择，另外很有可能还需要 (sex, country, region, age) 和 (sex, country, region, city, age) 这样的组合索引。

这样就会需要大量的索引。如果想尽可能重用索引而不是建立大量的组合索引，可以使用前面提到的 IN() 的技巧来避免同时需要 (sex, country, age) 和 (sex, country, region, age) 的索引。如果没有指定这个字段搜索，就需要定义一个全部国家列表，或者国家的全部地区列表，来确保索引前缀有同样的约束（组合所有国家、地区、性别将会是一个非常大的条件）。

这些索引将满足大部分最常见的搜索查询，但是如何为一些生僻的搜索条件（比如 has_pictures、eye_color、hair_color 和 education）来设计索引呢？这些列的选择性高、使用也不频繁，可以选择忽略它们，让 MySQL 多扫描一些额外的行即可。另一个可选的方法是在 age 列的前面加上这些列，在查询时使用前面提到过的 IN() 技术来处理搜索时没有指定这些列的场景。

你可能已经注意到了，我们一直将 age 列放在索引的最后面。age 列有什么特殊的地方吗？为什么要放在索引的最后？我们总是尽可能让 MySQL 使用更多的索引列，因为查询只能使用索引的最左前缀，直到遇到第一个范围条件列。前面提到的列在 WHERE 子句中都是等于条件，但是 age 列则多半是范围查询（例如查找年龄在 18 ~ 25 岁之间的人）。

当然，也可以使用 IN() 来代替范围查询，例如年龄条件改写为 IN(18, 19, 20, 21, 22, 23, 24, 25)，但不是所有的范围查询都可以转换。这里描述的基本原则是，尽可能将需要做范围查询的列放到索引的后面，以便优化器能使用尽可能多的索引列。

前面提到可以在索引中加入更多的列，并通过 IN() 的方式覆盖那些不在 WHERE 子句中的列。但这种技巧也不能滥用，否则可能会带来麻烦。因为每额外增加一个 IN() 条件，优化器需要做的组合都将以指数形式增加，最终可能会极大 地降低查询性能。考虑下面的 WHERE 子句：

```
WHERE eye_color   IN('brown','blue','hazel')
   AND hair_color IN('black','red','blonde','brown')
   AND sex        IN('M','F')
```

优化器则会转化成 4 × 3 × 2 = 24 种组合，执行计划需要检查 WHERE 子句中所有的 24 种

组合。对于 MySQL 来说，24 种组合并不是很夸张，但如果组合数达到上千个则需要特别小心。老版本的 MySQL 在 IN() 组合条件过多的时候会有很多问题。查询优化可能需要花很多时间，并消耗大量的内存。新版本的 MySQL 在组合数超过一定数量后就不再进行执行计划评估了，这可能会导致 MySQL 不能很好地利用索引。

5.4.2 避免多个范围条件

192

假设我们有一个 last_online 列并希望通过下面的查询显示在过去几周上线过的用户：

```
WHERE  eye_color   IN('brown','blue','hazel')
   AND hair_color  IN('black','red','blonde','brown')
   AND sex         IN('M','F')
   AND last_online > DATE_SUB(NOW(), INTERVAL 7 DAY)
   AND age         BETWEEN 18 AND 25
```

什么是范围条件？

从 EXPLAIN 的输出很难区分 MySQL 是要查询范围值，还是查询列表值。EXPLAIN 使用同样的词“range”来描述这两种情况。例如，从 type 列来看，MySQL 会把下面这种查询当作是“range”类型：

```
mysql> EXPLAIN SELECT actor_id FROM sakila.actor
    -> WHERE actor_id > 45\G
************************* 1. row *************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: range
```

但是下面这条查询呢？

```
mysql> EXPLAIN SELECT actor_id FROM sakila.actor
    -> WHERE actor_id IN(1, 4, 99)\G
************************* 1. row *************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: range
```

从 EXPLAIN 的结果是无法区分这两者的，但可以从值的范围和多个等于条件来得出不同。在我们看来，第二个查询就是多个等值条件查询。

我们不是挑剔：这两种访问效率是不同的。对于范围条件查询，MySQL 无法再使用范围列后面的其他索引列了，但是对于“多个等值条件查询”则没有这个限制。

这个查询有一个问题：它有两个范围条件，last_online 列和 age 列，MySQL 可以使用 last_online 列索引或者 age 列索引，但无法同时使用它们。

如果条件中只有 last_online 而没有 age，那么我们可能考虑在索引的后面加上 last_online 列。这里考虑如果我们无法把 age 字段转换为一个 IN() 的列表，并且仍要求对于同时有 last_online 和 age 这两个维度的范围查询的速度很快，那该怎么办？答案是，很遗憾没有一个直接的办法能够解决这个问题。但是我们能够将其中的一个范围查询转换为一个简单的等值比较。为了实现这一点，我们需要事先计算好一个 active 列，这个
193 字段由定时任务来维护。当用户每次登录时，将对应值设置为 1，并且将过去连续七天未曾登录的用户的值设置为 0。

这个方法可以让 MySQL 使用（active, sex, country, age）索引。active 列并不是完全精确的，但是对于这类查询来说，对精度的要求也没有那么高。如果需要精确数据，可以把 last_online 列放到 WHERE 子句，但不加入到索引中。这和本章前面通过计算 URL 哈希值来实现 URL 的快速查找类似。所以这个查询条件没法使用任何索引，但因为这个条件的过滤性不高，即使在索引中加入该列也没有太大的帮助。换个角度来说，缺乏合适的索引对该查询的影响也不明显。

到目前为止，我们可以看到：如果用户希望同时看到活跃和不活跃的用户，可以在查询中使用 IN() 列表。我们已经加入了很多这样的列表，但另外一个可选的方案就只能是为不同的组合列创建单独的索引。至少需要建立如下的索引：(active, sex, country, age)，(active, country, age)，(sex, country, age) 和 (country, age)。这些索引对某个具体的查询来说可能都是更优化的，但是考虑到索引的维护和额外的空间占用的代价，这个可选方案就不是一个好策略了。

在这个案例中，优化器的特性是影响索引策略的一个很重要的因素。如果未来版本的 MySQL 能够实现松散索引扫描，就能在一个索引上使用多个范围条件，那也就不需要为上面考虑的这类查询使用 IN() 列表了。

5.4.3 优化排序

在这个学习案例中，最后要介绍的是排序。使用文件排序对小数据集是很快的，但如果一个查询匹配的结果有上百万行的话会怎样？例如如果 WHERE 子句只有 sex 列，如何排序？

对于那些选择性非常低的列，可以增加一些特殊的索引来做排序。例如，可以创建（sex, rating）索引用于下面的查询：

```
mysql> SELECT <cols> FROM profiles WHERE sex='M' ORDER BY rating LIMIT 10;
```

这个查询同时使用了 ORDER BY 和 LIMIT，如果没有索引的话会很慢。

即使有索引，如果用户界面上需要翻页，并且翻页翻到比较靠后时查询也可能非常慢。下面这个查询就通过 ORDER BY 和 LIMIT 偏移量的组合翻页到很后面的时候 ：

```
mysql> SELECT <cols> FROM profiles WHERE sex='M' ORDER BY rating LIMIT 100000, 10;
```

无论如何创建索引，这种查询都是个严重的问题。因为随着偏移量的增加，MySQL 需要花费大量的时间来扫描需要丢弃的数据。反范式化、预先计算和缓存可能是解决这类查询的仅有策略。一个更好的办法是限制用户能够翻页的数量，实际上这对用户体验的 194
影响不大，因为用户很少会真正在乎搜索结果的第 10 000 页。

优化这类索引的另一个比较好的策略是使用延迟关联，通过使用覆盖索引查询返回需要的主键，再根据这些主键关联原表获得需要的行。这可以减少 MySQL 扫描那些需要丢弃的行数。下面这个查询显示了如何高效地使用（sex, rating）索引进行排序和分页 ：

```
mysql> SELECT <cols> FROM profiles INNER JOIN (
    ->    SELECT <primary key cols> FROM profiles
    ->    WHERE x.sex='M' ORDER BY rating LIMIT 100000, 10
    -> ) AS x USING(<primary key cols>);
```

5.5 维护索引和表

即使用正确的类型创建了表并加上了合适的索引，工作也没有结束 ：还需要维护表和索引来确保它们都正常工作。维护表有三个主要的目的 ：找到并修复损坏的表，维护准确的索引统计信息，减少碎片。

5.5.1 找到并修复损坏的表

表损坏（corruption）是很糟糕的事情。对于 MyISAM 存储引擎，表损坏通常是系统崩溃导致的。其他的引擎也会由于硬件问题、MySQL 本身的缺陷或者操作系统的问题导致索引损坏。

损坏的索引会导致查询返回错误的结果或者莫须有的主键冲突等问题，严重时甚至还会导致数据库的崩溃。如果你遇到了古怪的问题——例如一些不应该发生的错误——可以尝试运行 CHECK TABLE 来检查是否发生了表损坏（注意有些存储引擎不支持该命令 ；而有些引擎则支持以不同的选项来控制完全检查表的方式）。CHECK TABLE 通常能够找出大多数的表和索引的错误。

可以使用 REPAIR TABLE 命令来修复损坏的表，但同样不是所有的存储引擎都支持该命令。如果存储引擎不支持，也可通过一个不做任何操作（no-op）的 ALTER 操作来重建表，例如修改表的存储引擎为当前的引擎。下面是一个针对 InnoDB 表的例子：

```
mysql> ALTER TABLE innodb_tbl ENGINE=INNODB;
```

此外，也可以使用一些存储引擎相关的离线工具，例如 *myisamchk*；或者将数据导出一份，然后再重新导入。不过，如果损坏的是系统区域，或者是表的“行数据”区域，而不是索引，那么上面的办法就没有用了。在这种情况下，可以从备份中恢复表，或者尝试从损坏的数据文件中尽可能地恢复数据。

195 如果 InnoDB 引擎的表出现了损坏，那么一定是发生了严重的错误，需要立刻调查一下原因。InnoDB 一般不会出现损坏。InnoDB 的设计保证了它并不容易被损坏。如果发生损坏，一般要么是数据库的硬件问题例如内存或者磁盘问题（有可能），要么是由于数据库管理员的错误例如在 MySQL 外部操作了数据文件（有可能），抑或是 InnoDB 本身的缺陷（不太可能）。常见的类似错误通常是由于尝试使用 *rsync* 备份 InnoDB 导致的。不存在什么查询能够让 InnoDB 表损坏，也不用担心暗处有“陷阱”。如果某条查询导致 InnoDB 数据的损坏，那一定是遇到了 bug，而不是查询的问题。

如果遇到数据损坏，最重要的是找出是什么导致了损坏，而不只是简单地修复，否则很有可能还会不断地损坏。可以通过设置 innodb_force_recovery 参数进入 InnoDB 的强制恢复模式来修复数据，更多细节可以参考 MySQL 手册。另外，还可以使用开源的 InnoDB 数据恢复工具箱 (InnoDB Data Recovery Toolkit) 直接从 InnoDB 数据文件恢复出数据（下载地址：*http://www.percona.com/software/mysql-innodb-data-recovery-tools/*）。

5.5.2 更新索引统计信息

MySQL 的查询优化器会通过两个 API 来了解存储引擎的索引值的分布信息，以决定如何使用索引。第一个 API 是 records_in_range()，通过向存储引擎传入两个边界值获取在这个范围大概有多少条记录。对于某些存储引擎，该接口返回精确值，例如 MyISAM；但对于另一些存储引擎则是一个估算值，例如 InnoDB。

第二个 API 是 info()，该接口返回各种类型的数据，包括索引的基数（每个键值有多少条记录）。

如果存储引擎向优化器提供的扫描行数信息是不准确的数据，或者执行计划本身太复杂以致无法准确地获取各个阶段匹配的行数，那么优化器会使用索引统计信息来估算扫描行数。MySQL 优化器使用的是基于成本的模型，而衡量成本的主要指标就是一个查询需要扫描多少行。如果表没有统计信息，或者统计信息不准确，优化器就很有可能做出

错误的决定。可以通过运行 ANALYZE TABLE 来重新生成统计信息解决这个问题。

每种存储引擎实现索引统计信息的方式不同，所以需要进行 ANALYZE TABLE 的频率也因不同的引擎而不同，每次运行的成本也不同：

- Memory 引擎根本不存储索引统计信息。
- MyISAM 将索引统计信息存储在磁盘中，ANALYZE TABLE 需要进行一次全索引扫描来计算索引基数。在整个过程中需要锁表。
- 直到 MySQL 5.5 版本，InnoDB 也不在磁盘存储索引统计信息，而是通过随机的索引访问进行评估并将其存储在内存中。 196

可以使用 SHOW INDEX FROM 命令来查看索引的基数（Cardinality）。例如：

```
mysql> SHOW INDEX FROM sakila.actor\G
*************************** 1. row ***************************
       Table: actor
  Non_unique: 0
    Key_name: PRIMARY
Seq_in_index: 1
 Column_name: actor_id
   Collation: A
 Cardinality: 200
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
*************************** 2. row ***************************
       Table: actor
  Non_unique: 1
    Key_name: idx_actor_last_name
Seq_in_index: 1
 Column_name: last_name
   Collation: A
 Cardinality: 200
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
```

这个命令输出了很多关于索引的信息，在 MySQL 手册中对上面每个字段的含义都有详细的解释。这里需要特别提及的是索引列的基数（Cardinality），其显示了存储引擎估算索引列有多少个不同的取值。在 MySQL 5.0 和更新的版本中，还可以通过 INFORMATION_SCHEMA.STATISTICS 表很方便地查询到这些信息。例如基于 INFORMATION_SCHEMA 的表，可以编写一个查询给出当前选择性比较低的索引。需要注意的是，如果服务器上的库表非常多，则从这里获取元数据的速度可能会非常慢，而且会给 MySQL 带来额外的压力。

InnoDB 的统计信息值得深入研究。InnoDB 引擎通过抽样的方式来计算统计信息，首先随机地读取少量的索引页面，然后以此为样本计算索引的统计信息。在老的 InnoDB 版本中，样本页面数是 8，新版本的 InnoDB 可以通过参数 innodb_stats_sample_pages 来设置样本页的数量。设置更大的值，理论上来说可以帮助生成更准确的索引信息，特别是对于某些超大的数据表来说，但具体设置多大合适依赖于具体的环境。

197 InnoDB 会在表首次打开，或者执行 ANALYZE TABLE，抑或表的大小发生非常大的变化（大小变化超过十六分之一或者新插入了 20 亿行都会触发）的时候计算索引的统计信息。

InnoDB 在打开某些 INFORMATION_SCHEMA 表，或者使用 SHOW TABLE STATUS 和 SHOW INDEX，抑或在 MySQL 客户端开启自动补全功能的时候都会触发索引统计信息的更新。如果服务器上有大量的数据，这可能就是个很严重的问题，尤其是当 I/O 比较慢的时候。客户端或者监控程序触发索引信息采样更新时可能会导致大量的锁，并给服务器带来很多的额外压力，这会让用户因为启动时间漫长而沮丧。只要 SHOW INDEX 查看索引统计信息，就一定会触发统计信息的更新。可以关闭 innodb_stats_on_metadata 参数来避免上面提到的问题。

如果使用 Percona 版本，使用的就是 XtraDB 引擎而不是原生的 InnoDB 引擎，那么可以通过 innodb_stats_auto_update 参数来禁止通过自动采样的方式更新索引统计信息，这时需要手动执行 ANALYZE TABLE 命令来更新统计信息。如果某些查询执行计划很不稳定的话，可以用该办法固化查询计划。我们当初引入这个参数也正是为了解决一些客户的这种问题。

如果想要更稳定的执行计划，并在系统重启后更快地生成这些统计信息，那么可以使用系统表来持久化这些索引统计信息。甚至还可以在不同的机器间迁移索引统计信息，这样新环境启动时就无须再收集这些数据。在 Percona 5.1 版本和官方的 5.6 版本都已经加入这个特性。在 Percona 版本中通过 innodb_use_sys_stats_table 参数可以启用该特性，官方 5.6 版本则通过 innodb_analyze_is_persistent 参数控制。

一旦关闭索引统计信息的自动更新，那么就需要周期性地使用 ANALYZE TABLE 来手动更新。否则，索引统计信息就会永远不变。如果数据分布发生大的变化，可能会出现一些很糟糕的执行计划。

5.5.3 减少索引和数据的碎片

B-Tree 索引可能会碎片化，这会降低查询的效率。碎片化的索引可能会以很差或者无序的方式存储在磁盘上。

根据设计，B-Tree 需要随机磁盘访问才能定位到叶子页，所以随机访问是不可避免的。然而，如果叶子页在物理分布上是顺序且紧密的，那么查询的性能就会更好。否则，对于范围查询、索引覆盖扫描等操作来说，速度可能会降低很多倍；对于索引覆盖扫描这一点更加明显。

表的数据存储也可能碎片化。然而，数据存储的碎片化比索引更加复杂。有三种类型的数据碎片。

行碎片（Row fragmentation） 198

这种碎片指的是数据行被存储为多个地方的多个片段中。即使查询只从索引中访问一行记录，行碎片也会导致性能下降。

行间碎片（Intra-row fragmentation）

行间碎片是指逻辑上顺序的页，或者行在磁盘上不是顺序存储的。行间碎片对诸如全表扫描和聚簇索引扫描之类的操作有很大的影响，因为这些操作原本能够从磁盘上顺序存储的数据中获益。

剩余空间碎片（Free space fragmentation）

剩余空间碎片是指数据页中有大量的空余空间。这会导致服务器读取大量不需要的数据，从而造成浪费。

对于 MyISAM 表，这三类碎片化都可能发生。但 InnoDB 不会出现短小的行碎片；InnoDB 会移动短小的行并重写到一个片段中。

可以通过执行 `OPTIMIZE TABLE` 或者导出再导入的方式来重新整理数据。这对多数存储引擎都是有效的。对于一些存储引擎如 MyISAM，可以通过排序算法重建索引的方式来消除碎片。老版本的 InnoDB 没有什么消除碎片化的方法。不过最新版本 InnoDB 新增了“在线”添加和删除索引的功能，可以通过先删除，然后再重新创建索引的方式来消除索引的碎片化。

对于那些不支持 `OPTIMIZE TABLE` 的存储引擎，可以通过一个不做任何操作（no-op）的 `ALTER TABLE` 操作来重建表。只需要将表的存储引擎修改为当前的引擎即可：

```
mysql> ALTER TABLE <table> ENGINE=<engine>;
```

对于开启了 `expand_fast_index_creation` 参数的 Percona Server，按这种方式重建表，则会同时消除表和索引的碎片化。但对于标准版本的 MySQL 则只会消除表（实际上是聚簇索引）的碎片化。可用先删除所有索引，然后重建表，最后重新创建索引的方式模拟 Percona Server 的这个功能。

应该通过一些实际测量而不是随意假设来确定是否需要消除索引和表的碎片化。Percona

的 XtraBackup 有个 --*stats* 参数以非备份的方式运行，而只是打印索引和表的统计情况，包括页中的数据量和空余空间。这可以用来确定数据的碎片化程度。另外也要考虑数据是否已经达到稳定状态，如果你进行碎片整理将数据压缩到一起，可能反而会导致后续的更新操作触发一系列的页分裂和重组，这会对性能造成不良的影响（直到数据再次达到新的稳定状态）。

199

5.6 总结

通过本章可以看到，索引是一个非常复杂的话题！MySQL 和存储引擎访问数据的方式，加上索引的特性，使得索引成为一个影响数据访问的有力而灵活的工作（无论数据是在磁盘中还是在内存中）。

在 MySQL 中，大多数情况下都会使用 B-Tree 索引。其他类型的索引大多只适用于特殊的目的。如果在合适的场景中使用索引，将大大提高查询的响应时间。本章将不再介绍更多这方面的内容了，最后值得总的回顾一下这些特性以及如何使用 B-Tree 索引。

在选择索引和编写利用这些索引的查询时，有如下三个原则始终需要记住：

1. 单行访问是很慢的。特别是在机械硬盘存储中（SSD 的随机 I/O 要快很多，不过这一点仍然成立）。如果服务器从存储中读取一个数据块只是为了获取其中一行，那么就浪费了很多工作。最好读取的块中能包含尽可能多所需要的行。使用索引可以创建位置引用以提升效率。
2. 按顺序访问范围数据是很快的，这有两个原因。第一，顺序 I/O 不需要多次磁盘寻道，所以比随机 I/O 要快很多（特别是对机械硬盘）。第二，如果服务器能够按需要顺序读取数据，那么就不再需要额外的排序操作，并且 `GROUP BY` 查询也无须再做排序和将行按组进行聚合计算了。
3. 索引覆盖查询是很快的。如果一个索引包含了查询需要的所有列，那么存储引擎就不需要再回表查找行。这避免了大量的单行访问，而上面的第 1 点已经写明单行访问是很慢的。

总的来说，编写查询语句时应该尽可能选择合适的索引以避免单行查找、尽可能地使用数据原生顺序从而避免额外的排序操作，并尽可能使用索引覆盖查询。这与本章开头提到的 Lahdenmaki 和 Leach 的书中的“三星”评价系统是一致的。

如果表上的每一个查询都能有一个完美的索引来满足当然是最好的。但不幸的是，要这么做有时可能需要创建大量的索引。还有一些时候对某些查询是不可能创建一个达到“三星”的索引的（例如查询要按照两个列排序，其中一个列正序，另一个列倒序）。这时必须有所取舍以创建最合适的索引，或者寻求替代策略（例如反范式化，或者提前计算

汇总表等）。

理解索引是如何工作的非常重要，应该根据这些理解来创建最合适的索引，而不是根据一些诸如“在多列索引中将选择性最高的列放在第一列”或“应该为 WHERE 子句中出现的所有列创建索引”之类的经验法则及其推论。

那如何判断一个系统创建的索引是合理的呢？一般来说，我们建议按响应时间来对查询进行分析。找出那些消耗最长时间的查询或者那些给服务器带来最大压力的查询（第 3 章中介绍了如何测量），然后检查这些查询的 schema、SQL 和索引结构，判断是否有查询扫描了太多的行，是否做了很多额外的排序或者使用了临时表，是否使用随机 I/O 访问数据，或者是有太多回表查询那些不在索引中的列的操作。 200

如果一个查询无法从所有可能的索引中获益，则应该看看是否可以创建一个更合适的索引来提升性能。如果不行，也可以看看是否可以重写该查询，将其转化成一个能够高效利用现有索引或者新创建索引的查询。这也是下一章要介绍的内容。

如果根据第 3 章介绍的基于响应时间的分析不能找出有问题的查询呢？是否可能有我们没有注意到的“很糟糕”的查询，需要一个更好的索引来获取更高的性能？一般来说，不可能。对于诊断时抓不到的查询，那就不是问题。但是，这个查询未来有可能会成为问题，因为应用程序、数据和负载都在变化。如果仍然想找到那些索引不是很合适的查询，并在它们成为问题前进行优化，则可以使用 *pt-query-digest* 的查询审查“review”功能，分析其 EXPLAIN 出来的执行计划。

第 6 章
查询性能优化

前面的章节我们介绍了如何设计最优的库表结构、如何建立最好的索引，这些对于高性能来说是必不可少的。但这些还不够——还需要合理的设计查询。如果查询写得很糟糕，即使库表结构再合理、索引再合适，也无法实现高性能。

查询优化、索引优化、库表结构优化需要齐头并进，一个不落。在获得编写 MySQL 查询的经验的同时，也将学习到如何为高效的查询设计表和索引。同样的，也可以学习到在优化库表结构时会影响到哪些类型的查询。这个过程需要时间，所以建议大家在学习后面章节的时候多回头看看这三章的内容。

本章将从查询设计的一些基本原则开始——这也是在发现查询效率不高的时候首先需要考虑的因素。然后会介绍一些更深的查询优化的技巧，并会介绍一些 MySQL 优化器内部的机制。我们将展示 MySQL 是如何执行查询的，你也将学会如何去改变一个查询的执行计划。最后，我们要看一下 MySQL 优化器在哪些方面做得还不够，并探索查询优化的模式，以帮助 MySQL 更有效地执行查询。

本章的目标是帮助大家更深刻地理解 MySQL 如何真正地执行查询，并明白高效和低效的原因何在，这样才能充分发挥 MySQL 的优势，并避开它的弱点。

6.1 为什么查询速度会慢

在尝试编写快速的查询之前，需要清楚一点，真正重要是响应时间。如果把查询看作是一个任务，那么它由一系列子任务组成，每个子任务都会消耗一定的时间。如果要优化查询，实际上要优化其子任务，要么消除其中一些子任务，要么减少子任务的执行次数，

要么让子任务运行得更快[注1]。

202 MySQL 在执行查询的时候有哪些子任务，哪些子任务运行的速度很慢？这里很难给出完整的列表，但如果按照第 3 章介绍的方法对查询进行剖析，就能看到查询所执行的子任务。通常来说，查询的生命周期大致可以按照顺序来看：从客户端，到服务器，然后在服务器上进行解析，生成执行计划，执行，并返回结果给客户端。其中“执行”可以认为是整个生命周期中最重要的阶段，这其中包括了大量为了检索数据到存储引擎的调用以及调用后的数据处理，包括排序、分组等。

在完成这些任务的时候，查询需要在不同的地方花费时间，包括网络，CPU 计算，生成统计信息和执行计划、锁等待（互斥等待）等操作，尤其是向底层存储引擎检索数据的调用操作，这些调用需要在内存操作、CPU 操作和内存不足时导致的 I/O 操作上消耗时间。根据存储引擎不同，可能还会产生大量的上下文切换以及系统调用。

在每一个消耗大量时间的查询案例中，我们都能看到一些不必要的额外操作、某些操作被额外地重复了很多次、某些操作执行得太慢等。优化查询的目的就是减少和消除这些操作所花费的时间。

再次申明一点，对于一个查询的全部生命周期，上面列的并不完整。这里我们只是想说明：了解查询的生命周期、清楚查询的时间消耗情况对于优化查询有很大的意义。有了这些概念，我们再一起来看看如何优化查询。

6.2 慢查询基础：优化数据访问

查询性能低下最基本的原因是访问的数据太多。某些查询可能不可避免地需要筛选大量数据，但这并不常见。大部分性能低下的查询都可以通过减少访问的数据量的方式进行优化。对于低效的查询，我们发现通过下面两个步骤来分析总是很有效：

1. 确认应用程序是否在检索大量超过需要的数据。这通常意味着访问了太多的行，但有时候也可能是访问了太多的列。
2. 确认 MySQL 服务器层是否在分析大量超过需要的数据行。

6.2.1 是否向数据库请求了不需要的数据

有些查询会请求超过实际需要的数据，然后这些多余的数据会被应用程序丢弃。这会给

注 1：有时候你可能还需要修改一些查询，减少这些查询对系统中运行的其他查询的影响。这种情况下，你是在减少一个查询的资源消耗，这我们在第 3 章已经讨论过。

MySQL 服务器带来额外的负担，并增加网络开销[注2]，另外也会消耗应用服务器的 CPU 和内存资源。

这里有一些典型案例：

查询不需要的记录 203

一个常见的错误是常常会误以为 MySQL 会只返回需要的数据，实际上 MySQL 却是先返回全部结果集再进行计算。我们经常会看到一些了解其他数据库系统的人会设计出这类应用程序。这些开发者习惯使用这样的技术，先使用 SELECT 语句查询大量的结果，然后获取前面的 *N* 行后关闭结果集（例如在新闻网站中取出 100 条记录，但是只是在页面上显示前面 10 条）。他们认为 MySQL 会执行查询，并只返回他们需要的 10 条数据，然后停止查询。实际情况是 MySQL 会查询出全部的结果集，客户端的应用程序会接收全部的结果集数据，然后抛弃其中大部分数据。最简单有效的解决方法就是在这样的查询后面加上 LIMIT。

多表关联时返回全部列

如果你想查询所有在电影 *Academy Dinosaur* 中出现的演员，千万不要按下面的写法编写查询：

```
mysql> SELECT * FROM sakila.actor
    -> INNER JOIN sakila.film_actor USING(actor_id)
    -> INNER JOIN sakila.film USING(film_id)
    -> WHERE sakila.film.title = 'Academy Dinosaur';
```

这将返回这三个表的全部数据列。正确的方式应该是像下面这样只取需要的列：

```
mysql> SELECT sakila.actor.* FROM sakila.actor...;
```

总是取出全部列

每次看到 SELECT * 的时候都需要用怀疑的眼光审视，是不是真的需要返回全部的列？很可能不是必需的。取出全部列，会让优化器无法完成索引覆盖扫描这类优化，还会为服务器带来额外的 I/O、内存和 CPU 的消耗。因此，一些 DBA 是严格禁止 SELECT * 的写法的，这样做有时候还能避免某些列被修改带来的问题。

当然，查询返回超过需要的数据也不总是坏事。在我们研究过的许多案例中，人们会告诉我们说这种有点浪费数据库资源的方式可以简化开发，因为能提高相同代码片段的复用性，如果清楚这样做的性能影响，那么这种做法也是值得考虑的。如果应用程序使用了某种缓存机制，或者有其他考虑，获取超过需要的数据也可能有其好处，但不要忘记这样做的代价是什么。获取并缓存所有的列的查询，相比多个独

注 2：如果应用服务器和数据库不在同一台主机上，网络开销就显得很明显了。即使是在同一台服务器上仍然会有数据传输的开销。

立的只获取部分列的查询可能就更有好处。

重复查询相同的数据

如果你不太小心，很容易出现这样的错误——不断地重复执行相同的查询，然后每次都返回完全相同的数据。例如，在用户评论的地方需要查询用户头像的 URL，那么用户多次评论的时候，可能就会反复查询这个数据。比较好的方案是，当初次查询的时候将这个数据缓存起来，需要的时候从缓存中取出，这样性能显然会更好。

204

6.2.2 MySQL 是否在扫描额外的记录

在确定查询只返回需要的数据以后，接下来应该看看查询为了返回结果是否扫描了过多的数据。对于 MySQL，最简单的衡量查询开销的三个指标如下：

- 响应时间
- 扫描的行数
- 返回的行数

没有哪个指标能够完美地衡量查询的开销，但它们大致反映了 MySQL 在内部执行查询时需要访问多少数据，并可以大概推算出查询运行的时间。这三个指标都会记录到 MySQL 的慢日志中，所以检查慢日志记录是找出扫描行数过多的查询的好办法。

响应时间

要记住，响应时间只是一个表面上的值。这样说可能看起来和前面关于响应时间的说法有矛盾？其实并不矛盾，响应时间仍然是最重要的指标，这有一点复杂，后面细细道来。

响应时间是两个部分之和：服务时间和排队时间。服务时间是指数据库处理这个查询真正花了多长时间。排队时间是指服务器因为等待某些资源而没有真正执行查询的时间——可能是等 I/O 操作完成，也可能是等待行锁，等等。遗憾的是，我们无法把响应时间细分到上面这些部分，除非有什么办法能够逐个测量上面这些消耗，不过很难做到。一般最常见和重要的等待是 I/O 和锁等待，但是实际情况更加复杂。

所以在不同类型的应用压力下，响应时间并没有什么一致的规律或者公式。诸如存储引擎的锁（表锁、行锁）、高并发资源竞争、硬件响应等诸多因素都会影响响应时间。所以，响应时间既可能是一个问题的结果也可能是一个问题的原因，不同案例情况不同，除非能够使用第 3 章的“单个查询问题还是服务器问题”一节介绍的技术来确定到底是因还是果。

当你看到一个查询的响应时间的时候，首先需要问问自己，这个响应时间是否是一个合理的值。实际上可以使用“快速上限估计”法来估算查询的响应时间，这是由 Tapio

Lahdenmaki 和 Mike Leach 编写的 *Relational Database Index Design and the Optimizers*（Wiley 出版社）一书提到的技术，限于篇幅，在这里不会详细展开。概括地说，了解这个查询需要哪些索引以及它的执行计划是什么，然后计算大概需要多少个顺序和随机 I/O，再用其乘以在具体硬件条件下一次 I/O 的消耗时间。最后把这些消耗都加起来，就可以获得一个大概参考值来判断当前响应时间是不是一个合理的值。 205

扫描的行数和返回的行数

分析查询时，查看该查询扫描的行数是非常有帮助的。这在一定程度上能够说明该查询找到需要的数据的效率高不高。

对于找出那些"糟糕"的查询，这个指标可能还不够完美，因为并不是所有的行的访问代价都是相同的。较短的行的访问速度更快，内存中的行也比磁盘中的行的访问速度要快得多。

理想情况下扫描的行数和返回的行数应该是相同的。但实际情况中这种"美事"并不多。例如在做一个关联查询时，服务器必须要扫描多行才能生成结果集中的一行。扫描的行数对返回的行数的比率通常很小，一般在 1:1 和 10:1 之间，不过有时候这个值也可能非常非常大。

扫描的行数和访问类型

在评估查询开销的时候，需要考虑一下从表中找到某一行数据的成本。MySQL 有好几种访问方式可以查找并返回一行结果。有些访问方式可能需要扫描很多行才能返回一行结果，也有些访问方式可能无须扫描就能返回结果。

在 `EXPLAIN` 语句中的 `type` 列反应了访问类型。访问类型有很多种，从全表扫描到索引扫描、范围扫描、唯一索引查询、常数引用等。这里列的这些，速度是从慢到快，扫描的行数也是从小到大。你不需要记住这些访问类型，但需要明白扫描表、扫描索引、范围访问和单值访问的概念。

如果查询没有办法找到合适的访问类型，那么解决的最好办法通常就是增加一个合适的索引，这也正是我们前一章讨论过的问题。现在应该明白为什么索引对于查询优化如此重要了。索引让 MySQL 以最高效、扫描行数最少的方式找到需要的记录。

例如，我们看看示例数据库 Sakila 中的一个查询案例：

```
mysql> SELECT * FROM sakila.film_actor WHERE film_id = 1;
```

这个查询将返回 10 行数据，从 `EXPLAIN` 的结果可以看到，MySQL 在索引 `idx_fk_film_`

id 上使用了 ref 访问类型来执行查询：

206

```
mysql> EXPLAIN SELECT * FROM sakila.film_actor WHERE film_id = 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: const
         rows: 10
        Extra:
```

EXPLAIN 的结果也显示 MySQL 预估需要访问 10 行数据。换句话说，查询优化器认为这种访问类型可以高效地完成查询。如果没有合适的索引会怎样呢？ MySQL 就不得不使用一种更糟糕的访问类型，下面我们来看看如果我们删除对应的索引再来运行这个查询：

```
mysql> ALTER TABLE sakila.film_actor DROP FOREIGN KEY fk_film_actor_film;
mysql> ALTER TABLE sakila.film_actor DROP KEY idx_fk_film_id;
mysql> EXPLAIN SELECT * FROM sakila.film_actor WHERE film_id = 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 5073
        Extra: Using where
```

正如我们预测的，访问类型变成了一个全表扫描（ALL），现在 MySQL 预估需要扫描 5073 条记录来完成这个查询。这里的“Using Where”表示 MySQL 将通过 WHERE 条件来筛选存储引擎返回的记录。

一般 MySQL 能够使用如下三种方式应用 WHERE 条件，从好到坏依次为：

- 在索引中使用 WHERE 条件来过滤不匹配的记录。这是在存储引擎层完成的。
- 使用索引覆盖扫描（在 Extra 列中出现了 Using index）来返回记录，直接从索引中过滤不需要的记录并返回命中的结果。这是在 MySQL 服务器层完成的，但无须再回表查询记录。
- 从数据表中返回数据，然后过滤不满足条件的记录（在 Extra 列中出现 Using Where）。这在 MySQL 服务器层完成，MySQL 需要先从数据表读出记录然后过滤。

上面这个例子说明了好的索引多么重要。好的索引可以让查询使用合适的访问类型，尽可能地只扫描需要的数据行。但也不是说增加索引就能让扫描的行数等于返回的行数。例如下面使用聚合函数 COUNT() 的查询[注3]：

```
mysql> SELECT actor_id, COUNT(*) FROM sakila.film_actor GROUP BY actor_id;
```

这个查询需要读取几千行数据，但是仅返回 200 行结果。没有什么索引能够让这样的查询减少需要扫描的行数。 207

不幸的是，MySQL 不会告诉我们生成结果实际上需要扫描多少行数据[注4]，而只会告诉我们生成结果时一共扫描了多少行数据。扫描的行数中的大部分都很可能是被 WHERE 条件过滤掉的，对最终的结果集并没有贡献。在上面的例子中，我们删除索引后，看到 MySQL 需要扫描所有记录然后根据 WHERE 条件过滤，最终只返回 10 行结果。理解一个查询需要扫描多少行和实际需要使用的行数需要先去理解这个查询背后的逻辑和思想。

如果发现查询需要扫描大量的数据但只返回少数的行，那么通常可以尝试下面的技巧去优化它：

- 使用索引覆盖扫描，把所有需要用的列都放到索引中，这样存储引擎无须回表获取对应行就可以返回结果了（在前面的章节中我们已经讨论过了）。
- 改变库表结构。例如使用单独的汇总表（这是我们在第 4 章中讨论的办法）。
- 重写这个复杂的查询，让 MySQL 优化器能够以更优化的方式执行这个查询（这是本章后续需要讨论的问题）。

6.3 重构查询的方式

在优化有问题的查询时，目标应该是找到一个更优的方法获得实际需要的结果——而不一定总是需要从 MySQL 获取一模一样的结果集。有时候，可以将查询转换一种写法让其返回一样的结果，但是性能更好。但也可以通过修改应用代码，用另一种方式完成查询，最终达到一样的目的。这一节我们将介绍如何通过这种方式来重构查询，并展示何时需要使用这样的技巧。

6.3.1 一个复杂查询还是多个简单查询

设计查询的时候一个需要考虑的重要问题是，是否需要将一个复杂的查询分成多个简单的查询。在传统实现中，总是强调需要数据库层完成尽可能多的工作，这样做的逻辑在于以前总是认为网络通信、查询解析和优化是一件代价很高的事情。

注 3： 更多内容请参考后面的“优化 COUNT() 查询”。
注 4： 例如关联查询结果返回的一条记录通常是由多条记录组成。——译者注

但是这样的想法对于 MySQL 并不适用，MySQL 从设计上让连接和断开连接都很轻量级，在返回一个小的查询结果方面很高效。现代的网络速度比以前要快很多，无论是带宽还是延迟。在某些版本的 MySQL 上，即使在一个通用服务器上，也能够运行每秒超过 10
208 万的查询，即使是一个千兆网卡也能轻松满足每秒超过 2000 次的查询。所以运行多个小查询现在已经不是大问题了。

MySQL 内部每秒能够扫描内存中上百万行数据，相比之下，MySQL 响应数据给客户端就慢得多了。在其他条件都相同的时候，使用尽可能少的查询当然是更好的。但是有时候，将一个大查询分解为多个小查询是很有必要的。别害怕这样做，好好衡量一下这样做是不是会减少工作量。稍后我们将通过本章的一个示例来展示这个技巧的优势。

不过，在应用设计的时候，如果一个查询能够胜任时还写成多个独立查询是不明智的。例如，我们看到有些应用对一个数据表做 10 次独立的查询来返回 10 行数据，每个查询返回一条结果，查询 10 次！

6.3.2 切分查询

有时候对于一个大查询我们需要"分而治之"，将大查询切分成小查询，每个查询功能完全一样，只完成一小部分，每次只返回一小部分查询结果。

删除旧的数据就是一个很好的例子。定期地清除大量数据时，如果用一个大的语句一次性完成的话，则可能需要一次锁住很多数据、占满整个事务日志、耗尽系统资源、阻塞很多小的但重要的查询。将一个大的 DELETE 语句切分成多个较小的查询可以尽可能小地影响 MySQL 性能，同时还可以减少 MySQL 复制的延迟。例如，我们需要每个月运行一次下面的查询：

```
mysql> DELETE FROM messages WHERE created < DATE_SUB(NOW(),INTERVAL 3 MONTH);
```

那么可以用类似下面的办法来完成同样的工作：

```
rows_affected = 0
do {
   rows_affected = do_query(
      "DELETE FROM messages WHERE created < DATE_SUB(NOW(),INTERVAL 3 MONTH)
      LIMIT 10000")
} while rows_affected > 0
```

一次删除一万行数据一般来说是一个比较高效而且对服务器[注5]影响也最小的做法（如果是事务型引擎，很多时候小事务能够更高效）。同时，需要注意的是，如果每次删除数据后，都暂停一会儿再做下一次删除，这样也可以将服务器上原本一次性的压力分散到

注 5： Percona Toolkit 中的 *pt-archiver* 工具就可以安全而简单地完成这类工作。

一个很长的时间段中，就可以大大降低对服务器的影响，还可以大大减少删除时锁的持有时间。

6.3.3 分解关联查询

209

很多高性能的应用都会对关联查询进行分解。简单地，可以对每一个表进行一次单表查询，然后将结果在应用程序中进行关联。例如，下面这个查询：

```
mysql> SELECT * FROM tag
    ->    JOIN tag_post ON tag_post.tag_id=tag.id
    ->    JOIN post ON tag_post.post_id=post.id
    -> WHERE tag.tag='mysql';
```

可以分解成下面这些查询来代替：

```
mysql> SELECT * FROM  tag WHERE tag='mysql';
mysql> SELECT * FROM  tag_post WHERE tag_id=1234;
mysql> SELECT * FROM  post WHERE  post.id in (123,456,567,9098,8904);
```

到底为什么要这样做？乍一看，这样做并没有什么好处，原本一条查询，这里却变成多条查询，返回的结果又是一模一样的。事实上，用分解关联查询的方式重构查询有如下的优势：

- 让缓存的效率更高。许多应用程序可以方便地缓存单表查询对应的结果对象。例如，上面查询中的 tag 已经被缓存了，那么应用就可以跳过第一个查询。再例如，应用中已经缓存了 ID 为 123、567、9098 的内容，那么第三个查询的 IN() 中就可以少几个 ID。另外，对 MySQL 的查询缓存来说[注6]，如果关联中的某个表发生了变化，那么就无法使用查询缓存了，而拆分后，如果某个表很少改变，那么基于该表的查询就可以重复利用查询缓存结果了。
- 将查询分解后，执行单个查询可以减少锁的竞争。
- 在应用层做关联，可以更容易对数据库进行拆分，更容易做到高性能和可扩展。
- 查询本身效率也可能会有所提升。这个例子中，使用 IN() 代替关联查询，可以让 MySQL 按照 ID 顺序进行查询，这可能比随机的关联要更高效。我们后续将详细介绍这点。
- 可以减少冗余记录的查询。在应用层做关联查询，意味着对于某条记录应用只需要查询一次，而在数据库中做关联查询，则可能需要重复地访问一部分数据。从这点看，这样的重构还可能会减少网络和内存的消耗。
- 更进一步，这样做相当于在应用中实现了哈希关联，而不是使用 MySQL 的嵌套循环关联。某些场景哈希关联的效率要高很多（本章后续我们将讨论这点）。

注 6： Query Cache。——译者注

在很多场景下，通过重构查询将关联放到应用程序中将会更加高效，这样的场景有很多，比如：当应用能够方便地缓存单个查询的结果的时候、当可以将数据分布到不同的MySQL服务器上的时候、当能够使用IN()的方式代替关联查询的时候、当查询中使用同一个数据表的时候。

210

6.4 查询执行的基础

当希望MySQL能够以更高的性能运行查询时，最好的办法就是弄清楚MySQL是如何优化和执行查询的。一旦理解这一点，很多查询优化工作实际上就是遵循一些原则让优化器能够按照预想的合理的方式运行。

换句话说，是时候回头看看我们前面讨论的内容了：MySQL执行一个查询的过程。根据图6-1，我们可以看到当向MySQL发送一个请求的时候，MySQL到底做了些什么：

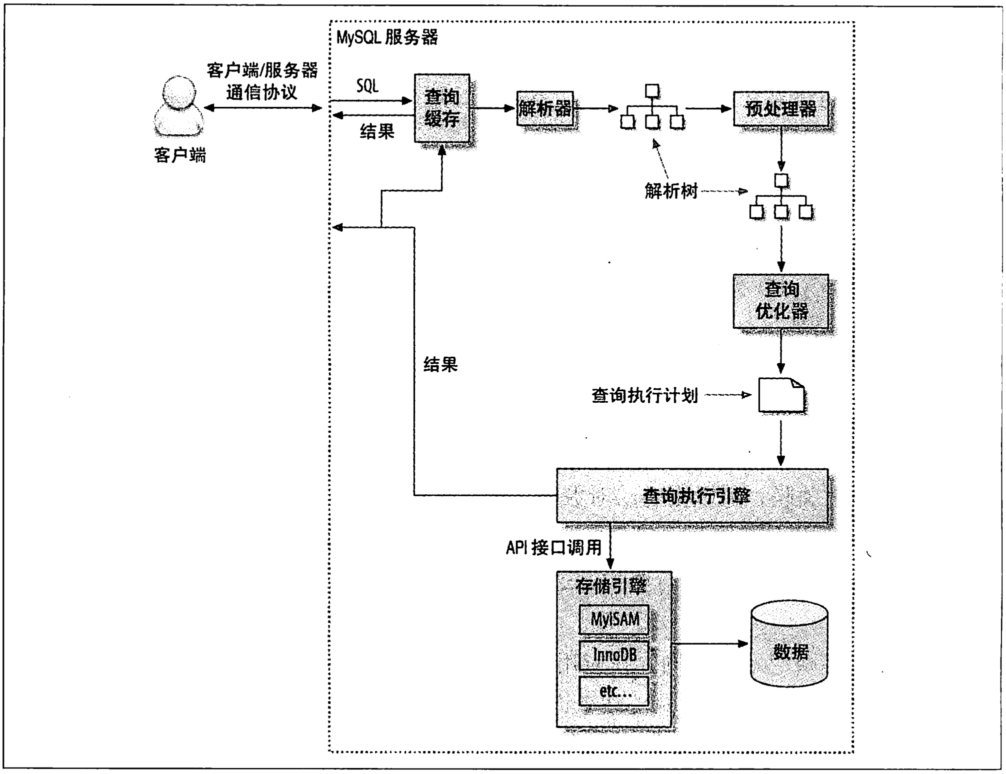


图6-1：查询执行路径

1. 客户端发送一条查询给服务器。
2. 服务器先检查查询缓存，如果命中了缓存，则立刻返回存储在缓存中的结果。否则进入下一阶段。
3. 服务器端进行 SQL 解析、预处理，再由优化器生成对应的执行计划。
4. MySQL 根据优化器生成的执行计划，调用存储引擎的 API 来执行查询。
5. 将结果返回给客户端。

上面的每一步都比想象的复杂，我们在后续章节中将继续讨论。我们会看到在每一个阶段查询处于何种状态。查询优化器是其中特别复杂也特别难理解的部分。还有很多的例外情况，例如，当查询使用绑定变量后，执行路径会有所不同，我们将在下一章讨论这点。

6.4.1 MySQL 客户端 / 服务器通信协议

一般来说，不需要去理解 MySQL 通信协议的内部实现细节，只需要大致理解通信协议是如何工作的。MySQL 客户端和服务器之间的通信协议是“半双工”的，这意味着，在任何一个时刻，要么是由服务器向客户端发送数据，要么是由客户端向服务器发送数据，这两个动作不能同时发生。所以，我们无法也无须将一个消息切成小块独立来发送。

这种协议让 MySQL 通信简单快速，但是也从很多地方限制了 MySQL。一个明显的限制是，这意味着没法进行流量控制。一旦一端开始发生消息，另一端要接收完整个消息才能响应它。这就像来回抛球的游戏：在任何时刻，只有一个人能控制球，而且只有控制球的人才能将球抛回去（发送消息）。

客户端用一个单独的数据包将查询传给服务器。这也是为什么当查询的语句很长的时候，参数 max_allowed_packet 就特别重要了[注7]。一旦客户端发送了请求，它能做的事情就只是等待结果了。

相反的，一般服务器响应给用户的数据通常很多，由多个数据包组成。当服务器开始响应客户端请求时，客户端必须完整地接收整个返回结果，而不能简单地只取前面几条结果，然后让服务器停止发送数据。这种情况下，客户端若接收完整的结果，然后取前面几条需要的结果，或者接收完几条结果后就“粗暴”地断开连接，都不是好主意。这也是在必要的时候一定要在查询中加上 LIMIT 限制的原因。 211

换一种方式解释这种行为：当客户端从服务器取数据时，看起来是一个拉数据的过程，但实际上是 MySQL 在向客户端推送数据的过程。客户端不断地接收从服务器推送的数据，客户端也没法让服务器停下来。客户端像是“从消防水管喝水”（这是一个术语）。

注 7： 如果查询太大，服务端会拒绝接收更多的数据并抛出相应错误。

多数连接MySQL的库函数都可以获得全部结果集并缓存到内存里，还可以逐行获取需
212 要的数据。默认一般是获得全部结果集并缓存到内存中。MySQL通常需要等所有的数据都已经发送给客户端才能释放这条查询所占用的资源，所以接收全部结果并缓存通常可以减少服务器的压力，让查询能够早点结束、早点释放相应的资源。

当使用多数连接MySQL的库函数从MySQL获取数据时，其结果看起来都像是从MySQL服务器获取数据，而实际上都是从这个库函数的缓存获取数据。多数情况下这没什么问题，但是如果需要返回一个很大的结果集的时候，这样做并不好，因为库函数会花很多时间和内存来存储所有的结果集。如果能够尽早开始处理这些结果集，就能大大减少内存的消耗，这种情况下可以不使用缓存来记录结果而是直接处理。这样做的缺点是，对于服务器来说，需要查询完成后才能释放资源，所以在和客户端交互的整个过程中，服务器的资源都是被这个查询所占用的[注8]。

我们看看当使用PHP的时候是什么情况。首先，下面是我们连接MySQL的通常写法：

```
<?php
$link   = mysql_connect('localhost', 'user', 'p4ssword');
$result = mysql_query('SELECT * FROM HUGE_TABLE', $link);
while ( $row = mysql_fetch_array($result) ) {
   // Do something with result
}
?>
```

这段代码看起来像是只有当你需要的时候，才通过循环从服务器端取出数据。而实际上，在上面的代码中，在调用mysql_query()的时候，PHP就已经将整个结果集缓存到内存中。下面的while循环只是从这个缓存中逐行取出数据，相反如果使用下面的查询，用mysql_unbuffered_query()代替mysql_query()，PHP则不会缓存结果：

```
<?php
$link   = mysql_connect('localhost', 'user', 'p4ssword');
$result = mysql_unbuffered_query('SELECT * FROM HUGE_TABLE', $link);
while ( $row = mysql_fetch_array($result) ) {
   // Do something with result
}
?>
```

不同的编程语言处理缓存的方式不同。例如，在Perl的DBD:mysql驱动中需要指定C连接库的mysql_use_result属性（默认是mysql_buffer_result）。下面是一个例子：

注8： 你可以使用SQL_BUFFER_RESULT，后面将再介绍这点。

```
#!/usr/bin/perl
use DBI;
my $dbh = DBI->connect('DBI:mysql:;host=localhost', 'user', 'p4ssword');
my $sth = $dbh->prepare('SELECT * FROM HUGE_TABLE', { mysql_use_result => 1 });
$sth->execute();
while ( my $row = $sth->fetchrow_array() ) {
   # Do something with result
}
```

注意到上面的 prepare() 调用指定了 mysql_use_result 属性为 1，所以应用将直接“使用”返回的结果集而不会将其缓存。也可以在连接 MySQL 的时候指定这个属性，这会让整个连接都使用不缓存的方式处理结果集： 213

```
my $dbh = DBI->connect('DBI:mysql:;mysql_use_result=1', 'user', 'p4ssword');
```

查询状态

对于一个 MySQL 连接，或者说一个线程，任何时刻都有一个状态，该状态表示了 MySQL 当前正在做什么。有很多种方式能查看当前的状态，最简单的是使用 SHOW FULL PROCESSLIST 命令（该命令返回结果中的 Command 列就表示当前的状态）。在一个查询的生命周期中，状态会变化很多次。MySQL 官方手册中对这些状态值的含义有最权威的解释，下面将这些状态列出来，并做一个简单的解释。

Sleep
: 线程正在等待客户端发送新的请求。

Query
: 线程正在执行查询或者正在将结果发送给客户端。

Locked
: 在 MySQL 服务器层，该线程正在等待表锁。在存储引擎级别实现的锁，例如 InnoDB 的行锁，并不会体现在线程状态中。对于 MyISAM 来说这是一个比较典型的状态，但在其他没有行锁的引擎中也经常会出现。

Analyzing and statistics
: 线程正在收集存储引擎的统计信息，并生成查询的执行计划。

Copying to tmp table [on disk]
: 线程正在执行查询，并且将其结果集都复制到一个临时表中，这种状态一般要么是在做 GROUP BY 操作，要么是文件排序操作，或者是 UNION 操作。如果这个状态后面还有“on disk”标记，那表示 MySQL 正在将一个内存临时表放到磁盘上。

Sorting result
: 线程正在对结果集进行排序。

Sending data

这表示多种情况：线程可能在多个状态之间传送数据，或者在生成结果集，或者在向客户端返回数据。

了解这些状态的基本含义非常有用，这可以让你很快地了解当前“谁正在持球”[注9]。在一个繁忙的服务器上，可能会看到大量的不正常的状态，例如 statistics 正占用大量的时间。这通常表示，某个地方有异常了，可以通过使用第 3 章的一些技巧来诊断到底是哪个环节出现了问题。

214

6.4.2 查询缓存[注10]

在解析一个查询语句之前，如果查询缓存是打开的，那么 MySQL 会优先检查这个查询是否命中查询缓存中的数据。这个检查是通过一个对大小写敏感的哈希查找实现的。查询和缓存中的查询即使只有一个字节不同，那也不会匹配缓存结果[注11]，这种情况下查询就会进入下一阶段的处理。

如果当前的查询恰好命中了查询缓存，那么在返回查询结果之前 MySQL 会检查一次用户权限。这仍然是无须解析查询 SQL 语句的，因为在查询缓存中已经存放了当前查询需要访问的表信息。如果权限没有问题，MySQL 会跳过所有其他阶段，直接从缓存中拿到结果并返回给客户端。这种情况下，查询不会被解析，不用生成执行计划，不会被执行。在第 7 章中的查询缓存一节，你将学习到更多细节。

6.4.3 查询优化处理

查询的生命周期的下一步是将一个 SQL 转换成一个执行计划，MySQL 再依照这个执行计划和存储引擎进行交互。这包括多个子阶段：解析 SQL、预处理、优化 SQL 执行计划。这个过程中任何错误（例如语法错误）都可能终止查询。这里不打算详细介绍 MySQL 内部实现，而只是选择性地介绍其中几个独立的部分，在实际执行中，这几部分可能一起执行也可能单独执行。我们的目的是帮助大家理解 MySQL 如何执行查询，以便写出更优秀的查询。

语法解析器和预处理

首先，MySQL 通过关键字将 SQL 语句进行解析，并生成一棵对应的“解析树”。MySQL 解析器将使用 MySQL 语法规则验证和解析查询。例如，它将验证是否使用错误

注 9： 回忆一下前面的客户端和服务器的“传球”比喻。——译者注

注 10： 这里是指 Query Cache。——译者注

注 11： Percona 版本的 MySQL 中提供了一个新的特性，可以在计算查询语句哈希值时，先将注释移除再算哈希值，这对于不同注释的相同查询可以命中相同的查询缓存结果。

的关键字，或者使用关键字的顺序是否正确等，再或者它还会验证引号是否能前后正确匹配。

预处理器则根据一些 MySQL 规则进一步检查解析树是否合法，例如，这里将检查数据表和数据列是否存在，还会解析名字和别名，看看它们是否有歧义。

下一步预处理器会验证权限。这通常很快，除非服务器上有非常多的权限配置。

查询优化器

215

现在语法树被认为是合法的了，并且由优化器将其转化成执行计划。一条查询可以有很多种执行方式，最后都返回相同的结果。优化器的作用就是找到这其中最好的执行计划。

MySQL 使用基于成本的优化器，它将尝试预测一个查询使用某种执行计划时的成本，并选择其中成本最小的一个。最初，成本的最小单位是随机读取一个 4K 数据页的成本，后来（成本计算公式）变得更加复杂，并且引入了一些“因子”来估算某些操作的代价，如当执行一次 WHERE 条件比较的成本。可以通过查询当前会话的 Last_query_cost 的值来得知 MySQL 计算的当前查询的成本。

```
mysql> SELECT SQL_NO_CACHE COUNT(*) FROM sakila.film_actor;
+----------+
| count(*) |
+----------+
|     5462 |
+----------+
mysql> SHOW STATUS LIKE 'Last_query_cost';
+-----------------+-------------+
| Variable_name   | Value       |
+-----------------+-------------+
| Last_query_cost | 1040.599000 |
+-----------------+-------------+
```

这个结果表示 MySQL 的优化器认为大概需要做 1 040 个数据页的随机查找才能完成上面的查询。这是根据一系列的统计信息计算得来的：每个表或者索引的页面个数、索引的基数（索引中不同值的数量）、索引和数据行的长度、索引分布情况。优化器在评估成本的时候并不考虑任何层面的缓存，它假设读取任何数据都需要一次磁盘 I/O。

有很多种原因会导致 MySQL 优化器选择错误的执行计划，如下所示：

- 统计信息不准确。MySQL 依赖存储引擎提供的统计信息来评估成本，但是有的存储引擎提供的信息是准确的，有的偏差可能非常大。例如，InnoDB 因为其 MVCC 的架构，并不能维护一个数据表的行数的精确统计信息。
- 执行计划中的成本估算不等同于实际执行的成本。所以即使统计信息精准，优化器给出的执行计划也可能不是最优的。例如有时候某个执行计划虽然需要读取更多的

页面，但是它的成本却更小。因为如果这些页面都是顺序读或者这些页面都已经在内存中的话，那么它的访问成本将很小。MySQL 层面并不知道哪些页面在内存中、哪些在磁盘上，所以查询实际执行过程中到底需要多少次物理 I/O 是无法得知的。

- MySQL 的最优可能和你想的最优不一样。你可能希望执行时间尽可能的短，但是 MySQL 只是基于其成本模型选择最优的执行计划，而有些时候这并不是最快的执行方式。所以，这里我们看到根据执行成本来选择执行计划并不是完美的模型。
- 216 MySQL 从不考虑其他并发执行的查询，这可能会影响到当前查询的速度。
- MySQL 也并不是任何时候都是基于成本的优化。有时也会基于一些固定的规则，例如，如果存在全文搜索的 MATCH() 子句，则在存在全文索引的时候就使用全文索引。即使有时候使用别的索引和 WHERE 条件可以远比这种方式要快，MySQL 也仍然会使用对应的全文索引。
- MySQL 不会考虑不受其控制的操作的成本，例如执行存储过程或者用户自定义函数的成本。
- 后面我们还会看到，优化器有时候无法去估算所有可能的执行计划，所以它可能错过实际上最优的执行计划。

MySQL 的查询优化器是一个非常复杂的部件，它使用了很多优化策略来生成一个最优的执行计划。优化策略可以简单地分为两种，一种是静态优化，一种是动态优化。静态优化可以直接对解析树进行分析，并完成优化。例如，优化器可以通过一些简单的代数变换将 WHERE 条件转换成另一种等价形式。静态优化不依赖于特别的数值，如 WHERE 条件中带入的一些常数等。静态优化在第一次完成后就一直有效，即使使用不同的参数重复执行查询也不会发生变化。可以认为这是一种"编译时优化"。

相反，动态优化则和查询的上下文有关，也可能和很多其他因素有关，例如 WHERE 条件中的取值、索引中条目对应的数据行数等。这需要在每次查询的时候都重新评估，可以认为这是"运行时优化"。

在执行语句和存储过程的时候，动态优化和静态优化的区别非常重要。MySQL 对查询的静态优化只需要做一次，但对查询的动态优化则在每次执行时都需要重新评估。有时候甚至在查询的执行过程中也会重新优化。[注 12]

下面是一些 MySQL 能够处理的优化类型：

重新定义关联表的顺序

数据表的关联并不总是按照在查询中指定的顺序进行。决定关联的顺序是优化器很

注 12：例如，在关联操作中，范围检查的执行计划会针对每一行重新评估索引。可以通过 EXPLAIN 执行计划中的 Extra 列是否有"range checked for each record"来确认这一点。该执行计划还会增加 select_full_range_join 这个服务器变量的值。

重要的一部分功能，本章后面将深入介绍这一点。

将外连接转化成内连接

并不是所有的 OUTER JOIN 语句都必须以外连接的方式执行。诸多因素，例如 WHERE 条件、库表结构都可能会让外连接等价于一个内连接。MySQL 能够识别这点并重写查询，让其可以调整关联顺序。

使用等价变换规则 217

MySQL 可以使用一些等价变换来简化并规范表达式。它可以合并和减少一些比较，还可以移除一些恒成立和一些恒不成立的判断。例如，(5=5 AND a>5) 将被改写为 a>5。类似的，如果有 (a<b AND b=c) AND a=5 则会改写为 b>5 AND b=c AND a=5。这些规则对于我们编写条件语句很有用，我们将在本章后续继续讨论。

优化 COUNT()、MIN() 和 MAX()

索引和列是否可为空通常可以帮助 MySQL 优化这类表达式。例如，要找到某一列的最小值，只需要查询对应 B-Tree 索引最左端的记录，MySQL 可以直接获取索引的第一行记录。在优化器生成执行计划的时候就可以利用这一点，在 B-Tree 索引中，优化器会将这个表达式作为一个常数对待。类似的，如果要查找一个最大值，也只需读取 B-Tree 索引的最后一条记录。如果 MySQL 使用了这种类型的优化，那么在 EXPLAIN 中就可以看到"Select tables optimized away"。从字面意思可以看出，它表示优化器已经从执行计划中移除了该表，并以一个常数取而代之。

类似的，没有任何 WHERE 条件的 COUNT(*) 查询通常也可以使用存储引擎提供的一些优化（例如，MyISAM 维护了一个变量来存放数据表的行数）。

预估并转化为常数表达式

当 MySQL 检测到一个表达式可以转化为常数的时候，就会一直把该表达式作为常数进行优化处理。例如，一个用户自定义变量在查询中没有发生变化时就可以转换为一个常数。数学表达式则是另一种典型的例子。

让人惊讶的是，在优化阶段，有时候甚至一个查询也能够转化为一个常数。一个例子是在索引列上执行 MIN() 函数。甚至是主键或者唯一键查找语句也可以转换为常数表达式。如果 WHERE 子句中使用了该类索引的常数条件，MySQL 可以在查询开始阶段就先查找到这些值，这样优化器就能够知道并转换为常数表达式。下面是一个例子：

```
mysql> EXPLAIN SELECT film.film_id, film_actor.actor_id
    -> FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    -> WHERE film.film_id = 1;
+----+-------------+------------+-------+----------------+-------+------+
| id | select_type | table      | type  | key            | ref   | rows |
+----+-------------+------------+-------+----------------+-------+------+
|  1 | SIMPLE      | film       | const | PRIMARY        | const |    1 |
|  1 | SIMPLE      | film_actor | ref   | idx_fk_film_id | const |   10 |
+----+-------------+------------+-------+----------------+-------+------+
```

MySQL 分两步来执行这个查询，也就是上面执行计划的两行输出。第一步先从 film 表找到需要的行。因为在 film_id 字段上有主键索引，所以 MySQL 优化器知道这只会返回一行数据，优化器在生成执行计划的时候，就已经通过索引信息知道将返回多少行数据。因为优化器已经明确知道有多少个值（WHERE 条件中的值）需要做索引查询，所以这里的表访问类型是 const。

218

在执行计划的第二步，MySQL 将第一步中返回的 film_id 列当作一个已知取值的列来处理。因为优化器清楚在第一步执行完成后，该值就会是明确的了。注意到正如第一步中一样，使用 film_actor 字段对表的访问类型也是 const。

另一种会看到常数条件的情况是通过等式将常数值从一个表传到另一个表，这可以通过 WHERE、USING 或者 ON 语句来限制某列取值为常数。在上面的例子中，因为使用了 USING 子句，优化器知道这也限制了 film_id 在整个查询过程中都始终是一个常量——因为它必须等于 WHERE 子句中的那个取值。

覆盖索引扫描

当索引中的列包含所有查询中需要使用的列的时候，MySQL 就可以使用索引返回需要的数据，而无须查询对应的数据行，在前面的章节中我们已经讨论过这点了。

子查询优化

MySQL 在某些情况下可以将子查询转换一种效率更高的形式，从而减少多个查询多次对数据进行访问。

提前终止查询

在发现已经满足查询需求的时候，MySQL 总是能够立刻终止查询。一个典型的例子就是当使用了 LIMIT 子句的时候。除此之外，MySQL 还有几类情况也会提前终止查询，例如发现了一个不成立的条件，这时 MySQL 可以立刻返回一个空结果。从下面的例子可以看到这一点：

```
mysql> EXPLAIN SELECT film.film_id FROM sakila.film WHERE film_id = -1;
+----+...+-----------------------------------------------------+
| id |...| Extra                                               |
+----+...+-----------------------------------------------------+
|  1 |...| Impossible WHERE noticed after reading const tables |
+----+...+-----------------------------------------------------+
```

从这个例子看到查询在优化阶段就已经终止。除此之外，MySQL 在执行过程中，如果发现某些特殊的条件，则会提前终止查询。当存储引擎需要检索“不同取值”或者判断存在性的时候，MySQL 都可以使用这类优化。例如，我们现在需要找到没有演员的所有电影[注13]：

注 13： 一部电影没有演员，是有点奇怪。不过在示例数据库 Sakila 中影片 *SLACKER LIAISONS* 没有任何演员，它的描述是“鲨鱼和见识过中国古代鳄鱼的学生的简短传说”。

```
mysql> SELECT film.film_id
    -> FROM sakila.film
    ->    LEFT OUTER JOIN sakila.film_actor USING(film_id)
    -> WHERE film_actor.film_id IS NULL;
```

这个查询将会过滤掉所有有演员的电影。每一部电影可能会有很多的演员，但是上面的查询一旦找到任何一个，就会停止并立刻判断下一部电影，因为只要有一名演员，那么 WHERE 条件则会过滤掉这类电影。类似这种“不同值 / 不存在”的优化一般可用于 DISTINCT、NOT EXIST() 或者 LEFT JOIN 类型的查询。 219

等值传播

如果两个列的值通过等式关联，那么 MySQL 能够把其中一个列的 WHERE 条件传递到另一列上。例如，我们看下面的查询：

```
mysql> SELECT film.film_id
    -> FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    -> WHERE film.film_id > 500;
```

因为这里使用了 film_id 字段进行等值关联，MySQL 知道这里的 WHERE 子句不仅适用于 film 表，而且对于 film_actor 表同样适用。如果使用的是其他的数据库管理系统，可能还需要手动通过一些条件来告知优化器这个 WHERE 条件适用于两个表，那么写法就会如下：

```
... WHERE film.film_id > 500 AND film_actor.film_id > 500
```

在 MySQL 中这是不必要的，这样写反而会让查询更难维护。

列表 IN() 的比较

在很多数据库系统中，IN() 完全等同于多个 OR 条件的子句，因为这两者是完全等价的。在 MySQL 中这点是不成立的，MySQL 将 IN() 列表中的数据先进行排序，然后通过二分查找的方式来确定列表中的值是否满足条件，这是一个 $O(\log n)$ 复杂度的操作，等价地转换成 OR 查询的复杂度为 $O(n)$，对于 IN() 列表中有大量取值的时候，MySQL 的处理速度将会更快。

上面列举的远不是 MySQL 优化器的全部，MySQL 还会做大量其他的优化，即使本章全部用来描述也会篇幅不足，但上面的这些例子已经足以让大家明白优化器的复杂性和智能性了。如果说从上面这段讨论中我们应该学到什么，那就是“不要自以为比优化器更聪明”。最终你可能会占点便宜，但是更有可能会使查询变得更加复杂而难以维护，而最终的收益却为零。让优化器按照它的方式工作就可以了。

当然，虽然优化器已经很智能了，但是有时候也无法给出最优的结果。有时候你可能比优化器更了解数据，例如，由于应用逻辑使得某些条件总是成立；还有时，优化器缺少某种功能特性，如哈希索引；再如前面提到的，从优化器的执行成本角度评估出来的最优执行计划，实际运行中可能比其他的执行计划更慢。

220 如果能够确认优化器给出的不是最佳选择，并且清楚背后的原理，那么也可以帮助优化器做进一步的优化。例如，可以在查询中添加 hint 提示，也可以重写查询，或者重新设计更优的库表结构，或者添加更合适的索引。

数据和索引的统计信息

重新回忆一下图 1-1，MySQL 架构由多个层次组成。在服务器层有查询优化器，却没有保存数据和索引的统计信息。统计信息由存储引擎实现，不同的存储引擎可能会存储不同的统计信息（也可以按照不同的格式存储统计信息）。某些引擎，例如 Archive 引擎，则根本就没有存储任何统计信息！

因为服务器层没有任何统计信息，所以 MySQL 查询优化器在生成查询的执行计划时，需要向存储引擎获取相应的统计信息。存储引擎则提供给优化器对应的统计信息，包括：每个表或者索引有多少个页面、每个表的每个索引的基数是多少、数据行和索引长度、索引的分布信息等。优化器根据这些信息来选择一个最优的执行计划。在后面的小节中我们将看到统计信息是如何影响优化器的。

MySQL 如何执行关联查询

MySQL 中"关联"[注14]一词所包含的意义比一般意义上理解的要更广泛。总的来说，MySQL 认为任何一个查询都是一次"关联"——并不仅仅是一个查询需要到两个表匹配才叫关联，所以在 MySQL 中，每一个查询，每一个片段（包括子查询，甚至基于单表的 SELECT）都可能是关联。

所以，理解 MySQL 如何执行关联查询至关重要。我们先来看一个 `UNION` 查询的例子。对于 `UNION` 查询，MySQL 先将一系列的单个查询结果放到一个临时表中，然后再重新读出临时表数据来完成 `UNION` 查询。在 MySQL 的概念中，每个查询都是一次关联，所以读取结果临时表也是一次关联。

当前 MySQL 关联执行的策略很简单：MySQL 对任何关联都执行嵌套循环关联操作，即 MySQL 先在一个表中循环取出单条数据，然后再嵌套循环到下一个表中寻找匹配的行，依次下去，直到找到所有表中匹配的行为止。然后根据各个表匹配的行，返回查询中需要的各个列。MySQL 会尝试在最后一个关联表中找到所有匹配的行，如果最后一个关

注 14： join。——译者注

联表无法找到更多的行以后，MySQL 返回到上一层次关联表，看是否能够找到更多的匹配记录，依此类推迭代执行。[注 15]

按照这样的方式查找第一个表记录，再嵌套查询下一个关联表，然后回溯到上一个表，在 MySQL 中是通过嵌套循环的方式实现——正如其名“嵌套循环关联”。请看下面的例子中的简单查询：

221

```
mysql> SELECT tbl1.col1, tbl2.col2
    -> FROM tbl1 INNER JOIN tbl2 USING(col3)
    -> WHERE tbl1.col1 IN(5,6);
```

假设 MySQL 按照查询中的表顺序进行关联操作，我们则可以用下面的伪代码表示 MySQL 将如何完成这个查询：

```
outer_iter = iterator over tbl1 where col1 IN(5,6)
outer_row  = outer_iter.next
while outer_row
   inner_iter = iterator over tbl2 where col3 = outer_row.col3
   inner_row  = inner_iter.next
   while inner_row
      output [ outer_row.col1, inner_row.col2 ]
      inner_row = inner_iter.next
   end
   outer_row = outer_iter.next
end
```

上面的执行计划对于单表查询和多表关联查询都适用，如果是一个单表查询，那么只需完成上面外层的基本操作。对于外连接上面的执行过程仍然适用。例如，我们将上面查询修改如下：

```
mysql> SELECT tbl1.col1, tbl2.col2
    -> FROM tbl1 LEFT OUTER JOIN tbl2 USING(col3)
    -> WHERE tbl1.col1 IN(5,6);
```

对应的伪代码如下，我们用黑体标示不同的部分：

```
outer_iter = iterator over tbl1 where col1 IN(5,6)
outer_row  = outer_iter.next
while outer_row
   inner_iter = iterator over tbl2 where col3 = outer_row.col3
   inner_row  = inner_iter.next
   if inner_row
      while inner_row
         output [ outer_row.col1, inner_row.col2 ]
         inner_row = inner_iter.next
      end
```

注 15：后面我们会看到 MySQL 查询执行过程并没有这么简单，MySQL 做了很多优化操作。

```
      else
         output [ outer_row.col1, NULL ]
      end
      outer_row = outer_iter.next
   end
```

另一种可视化查询执行计划的方法是根据优化器执行的路径绘制出对应的“泳道图”。如图 6-2 所示，绘制了前面示例中内连接的泳道图，请从左至右，从上至下地看这幅图。

222

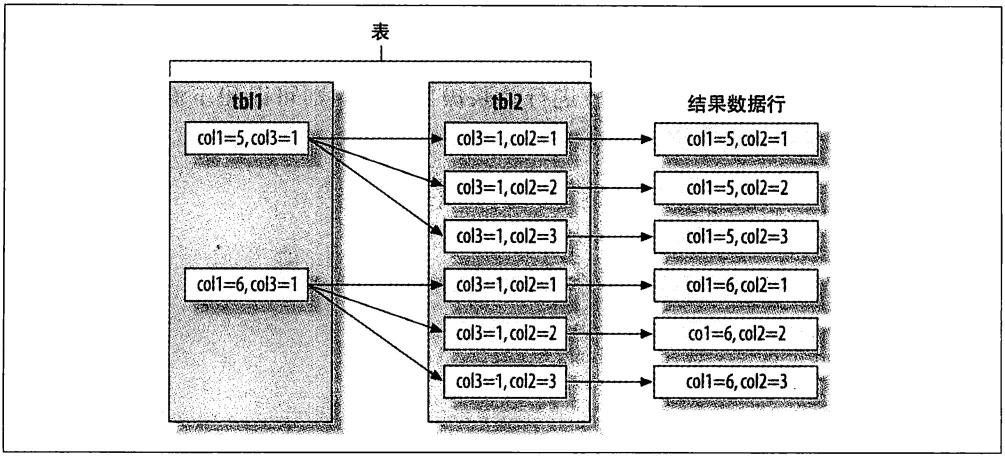


图6-2：通过泳道图展示MySQL如何完成关联查询

从本质上说，MySQL 对所有的类型的查询都以同样的方式运行。例如，MySQL 在 FROM 子句中遇到子查询时，先执行子查询并将其结果放到一个临时表中[注16]，然后将这个临时表当作一个普通表对待（正如其名“派生表”）。MySQL 在执行 UNION 查询时也使用类似的临时表，在遇到右外连接的时候，MySQL 将其改写成等价的左外连接。简而言之，当前版本的 MySQL 会将所有的查询类型都转换成类似的执行计划。[注17]

不过，不是所有的查询都可以转换成上面的形式。例如，全外连接就无法通过嵌套循环和回溯的方式完成，这时当发现关联表中没有找到任何匹配行的时候，则可能是因为关联是恰好从一个没有任何匹配的表开始。这大概也是 MySQL 并不支持全外连接的原因。还有些场景，虽然可以转换成嵌套循环的方式，但是效率却非常差，后面我们会看一个这样的例子。

注 16： MySQL 的临时表是没有任何索引的，在编写复杂的子查询和关联查询的时候需要注意这一点。这一点对 UNION 查询也一样。

注 17： 在 MySQL 5.6 和 MariaDB 中有了重大改变，这两个版本都引入了更加复杂的执行计划。

执行计划

和很多其他关系数据库不同，MySQL 并不会生成查询字节码来执行查询。MySQL 生成查询的一棵指令树，然后通过存储引擎执行完成这棵指令树并返回结果。最终的执行计划包含了重构查询的全部信息。如果对某个查询执行 EXPLAIN EXTENDED 后，再执行 SHOW WARNINGS，就可以看到重构出的查询[注18]。

任何多表查询都可以使用一棵树表示，例如，可以按照图 6-3 执行一个四表的关联操作。 223

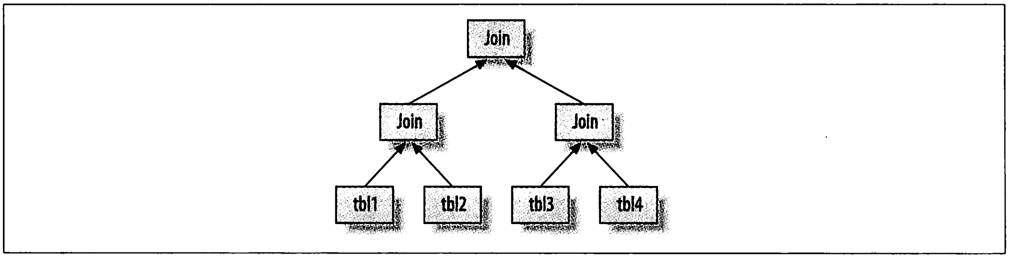


图6-3：多表关联的一种方式

在计算机科学中，这被称为一颗平衡树。但是，这并不是 MySQL 执行查询的方式。正如我们前面章节介绍的，MySQL 总是从一个表开始一直嵌套循环、回溯完成所有表关联。所以，MySQL 的执行计划总是如图 6-4 所示，是一棵左测深度优先的树。

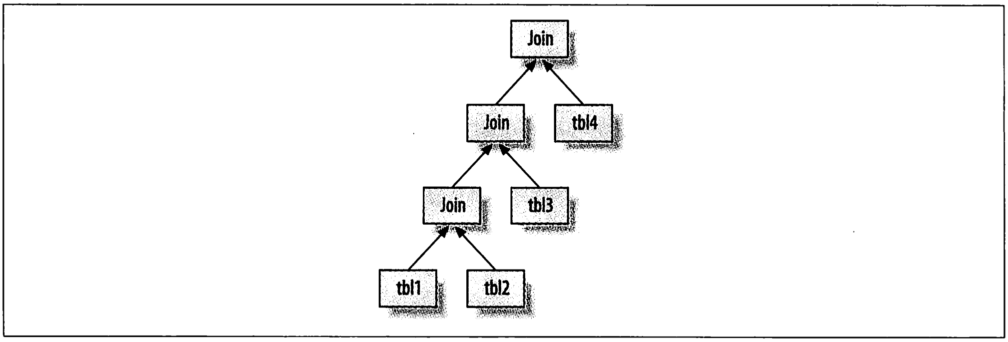


图6-4：MySQL如何实现多表关联

关联查询优化器

MySQL 优化器最重要的一部分就是关联查询优化，它决定了多个表关联时的顺序。通常多表关联的时候，可以有多种不同的关联顺序来获得相同的执行结果。关联查询优化

注 18： MySQL 根据执行计划生成输出。这和原查询有完全相同的语义，但是查询语句可能并不完全相同。

器则通过评估不同顺序时的成本来选择一个代价最小的关联顺序。

下面的查询可以通过不同顺序的关联最后都获得相同的结果：

224

```
mysql> SELECT film.film_id, film.title, film.release_year, actor.actor_id,
    ->    actor.first_name, actor.last_name
    ->    FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    ->    INNER JOIN sakila.actor USING(actor_id);
```

容易看出，可以通过一些不同的执行计划来完成上面的查询。例如，MySQL 可以从 film 表开始，使用 film_actor 表的索引 film_id 来查找对应的 actor_id 值，然后再根据 actor 表的主键找到对应的记录。Oracle 用户会用下面的术语描述："film 表作为驱动表先查找 file_actor 表，然后以此结果为驱动表再查找 actor 表"。这样做效率应该会不错，我们再使用 EXPLAIN 看看 MySQL 将如何执行这个查询：

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 200
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: PRIMARY,idx_fk_film_id
          key: PRIMARY
      key_len: 2
          ref: sakila.actor.actor_id
         rows: 1
        Extra: Using index
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.film_actor.film_id
         rows: 1
        Extra:
```

这和我们前面给出的执行计划完全不同。MySQL 从 actor 表开始（我们从上面的

EXPLAIN 结果的第一行输出可以看出这点），然后与我们前面的计划按照相反的顺序进行关联。这样是否效率更高呢？我们来看看，我们先使用 STRAIGHT_JOIN 关键字，按照我们之前的顺序执行，这里是对应的 EXPLAIN 输出结果：

```
mysql> EXPLAIN SELECT STRAIGHT_JOIN film.film_id...\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 951
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: PRIMARY,idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: sakila.film.film_id
         rows: 1
        Extra: Using index
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.film_actor.actor_id
         rows: 1
        Extra:
```

225

我们来分析一下为什么 MySQL 会将关联顺序倒转过来：可以看到，关联顺序倒转后的第一个关联表只需要扫描很少的行数[注19]。在两种关联顺序下，第二个和第三个关联表都是根据索引查询，速度都很快，不同的是需要扫描的索引项的数量是不同的：

- 将 film 表作为第一个关联表时，会找到 951 条记录，然后对 film_actor 和 actor 表进行嵌套循环查询。
- 如果 MySQL 选择首先扫描 actor 表，只会返回 200 条记录进行后面的嵌套循环查询。

注 19：严格来说，MySQL 并不根据读取的记录来选择最优的执行计划。实际上，MySQL 通过预估需要读取的数据页来选择，读取的数据页越少越好。不过读取的记录数通常能够很好地反映一个查询的成本。

换句话说，倒转的关联顺序会让查询进行更少的嵌套循环和回溯操作。为了验证优化器的选择是否正确，我们单独执行这两个查询，并且看看对应的 Last_query_cost 状态值。我们看到倒转的关联顺序的预估成本[注20]为 241，而原来的查询的预估成本为 1 154。

这个简单的例子主要想说明 MySQL 是如何选择合适的关联顺序来让查询执行的成本尽可能低的。重新定义关联的顺序是优化器非常重要的一部分功能。不过有的时候，优化
226 器给出的并不是最优的关联顺序。这时可以使用 STRAIGHT_JOIN 关键字重写查询，让优化器按照你认为的最优的关联顺序执行——不过老实说，人的判断很难那么精准。绝大多数时候，优化器做出的选择都比普通人的判断要更准确。

关联优化器会尝试在所有的关联顺序中选择一个成本最小的来生成执行计划树。如果可能，优化器会遍历每一个表然后逐个做嵌套循环计算每一棵可能的执行计划树的成本，最后返回一个最优的执行计划。

不过，糟糕的是，如果有超过 *n* 个表的关联，那么需要检查 *n* 的阶乘种关联顺序。我们称之为所有可能的执行计划的“搜索空间”，搜索空间的增长速度非常块——例如，若是 10 个表的关联，那么共有 3 628 800 种不同的关联顺序！当搜索空间非常大的时候，优化器不可能逐一评估每一种关联顺序的成本。这时，优化器选择使用“贪婪”搜索的方式查找“最优”的关联顺序。实际上，当需要关联的表超过 optimizer_search_depth 的限制的时候，就会选择“贪婪”搜索模式了（optimizer_search_depth 参数可以根据需要指定大小）。

在 MySQL 这些年的发展过程中，优化器积累了很多“启发式”的优化策略来加速执行计划的生成。绝大多数情况下，这都是有效的，但因为不会去计算每一种关联顺序的成本，所以偶尔也会选择一个不是最优的执行计划。

有时，各个查询的顺序并不能随意安排，这时关联优化器可以根据这些规则大大减少搜索空间，例如，左连接、相关子查询（后面我将继续讨论子查询）。这是因为，后面的表的查询需要依赖于前面表的查询结果。这种依赖关系通常可以帮助优化器大大减少需要扫描的执行计划数量。

排序优化

无论如何排序都是一个成本很高的操作，所以从性能角度考虑，应尽可能避免排序或者尽可能避免对大量数据进行排序。

在第 3 章中我们已经看到 MySQL 如何通过索引进行排序。当不能使用索引生成排序结

注 20：查询的 cost。——译者注

果的时候，MySQL 需要自己进行排序，如果数据量小则在内存中进行，如果数据量大则需要使用磁盘，不过 MySQL 将这个过程统一称为文件排序（*filesort*），即使完全是内存排序不需要任何磁盘文件时也是如此。

如果需要排序的数据量小于“排序缓冲区”，MySQL 使用内存进行“快速排序”操作。如果内存不够排序，那么 MySQL 会先将数据分块，对每个独立的块使用“快速排序”进行排序，并将各个块的排序结果存放在磁盘上，然后将各个排好序的块进行合并（merge），最后返回排序结果。

MySQL 有如下两种排序算法：

两次传输排序（旧版本使用）

读取行指针和需要排序的字段，对其进行排序，然后再根据排序结果读取所需要的数据行。

这需要进行两次数据传输，即需要从数据表中读取两次数据，第二次读取数据的时候，因为是读取排序列进行排序后的所有记录，这会产生大量的随机 I/O，所以两次数据传输的成本非常高。当使用的是 MyISAM 表的时候，成本可能会更高，因为 MyISAM 使用系统调用进行数据的读取（MyISAM 非常依赖操作系统对数据的缓 227
存）。不过这样做的优点是，在排序的时候存储尽可能少的数据，这就让“排序缓冲区”[注21]中可能容纳尽可能多的行数进行排序。

单次传输排序（新版本使用）

先读取查询所需要的所有列，然后再根据给定列进行排序，最后直接返回排序结果。这个算法只在 MySQL 4.1 和后续更新的版本才引入。因为不再需要从数据表中读取两次数据，对于 I/O 密集型的应用，这样做的效率高了很多。另外，相比两次传输排序，这个算法只需要一次顺序 I/O 读取所有的数据，而无须任何的随机 I/O。缺点是，如果需要返回的列非常多、非常大，会额外占用大量的空间，而这些列对排序操作本身来说是没有任何作用的。因为单条排序记录很大，所以可能会有更多的排序块需要合并。

很难说哪个算法效率更高，两种算法都有各自最好和最糟的场景。当查询需要所有列的总长度不超过参数 `max_length_for_sort_data` 时，MySQL 使用“单次传输排序”，可以通过调整这个参数来影响 MySQL 排序算法的选择。关于这个细节，可以参考第 8 章“文件排序优化”。

MySQL 在进行文件排序的时候需要使用的临时存储空间可能会比想象的要大得多。原因在于 MySQL 在排序时，对每一个排序记录都会分配一个足够长的定长空间来存放。

注 21： 内存。——译者注

这个定长空间必须足够长以容纳其中最长的字符串，例如，如果是 VARCHAR 列则需要分配其完整长度；如果使用 UTF-8 字符集，那么 MySQL 将会为每个字符预留三个字节。我们曾经在一个库表结构设计不合理的案例中看到，排序消耗的临时空间比磁盘上的原表要大很多倍。

在关联查询的时候如果需要排序，MySQL 会分两种情况来处理这样的文件排序。如果 ORDER BY 子句中的所有列都来自关联的第一个表，那么 MySQL 在关联处理第一个表的时候就进行文件排序。如果是这样，那么在 MySQL 的 EXPLAIN 结果中可以看到 Extra 字段会有"Using filesort"。除此之外的所有情况，MySQL 都会先将关联的结果存放到一个临时表中，然后在所有的关联都结束后，再进行文件排序。这种情况下，在 MySQL 的 EXPLAIN 结果的 Extra 字段可以看到"Using temporary; Using filesort"。如果查询中有 LIMIT 的话，LIMIT 也会在排序之后应用，所以即使需要返回较少的数据，临时表和需要排序的数据量仍然会非常大。

MySQL 5.6 在这里做了很多重要的改进。当只需要返回部分排序结果的时候，例如使用了 LIMIT 子句，MySQL 不再对所有的结果进行排序，而是根据实际情况，选择抛弃不满足条件的结果，然后再进行排序。

228 6.4.4 查询执行引擎

在解析和优化阶段，MySQL 将生成查询对应的执行计划，MySQL 的查询执行引擎则根据这个执行计划来完成整个查询。这里执行计划是一个数据结构，而不是和很多其他的关系型数据库那样会生成对应的字节码。

相对于查询优化阶段，查询执行阶段不是那么复杂：MySQL 只是简单地根据执行计划给出的指令逐步执行。在根据执行计划逐步执行的过程中，有大量的操作需要通过调用存储引擎实现的接口来完成，这些接口也就是我们称为"*handler API*"的接口。查询中的每一个表由一个 handler 的实例表示。前面我们有意忽略了这点，实际上，MySQL 在优化阶段就为每个表创建了一个 handler 实例，优化器根据这些实例的接口可以获取表的相关信息，包括表的所有列名、索引统计信息，等等。

存储引擎接口有着非常丰富的功能，但是底层接口却只有几十个，这些接口像"搭积木"一样能够完成查询的大部分操作。例如，有一个查询某个索引的第一行的接口，再有一个查询某个索引条目的下一个条目的功能，有了这两个功能我们就可以完成全索引扫描的操作了。这种简单的接口模式，让 MySQL 的存储引擎插件式架构成为可能，但是正如前面的讨论，也给优化器带来了一定的限制。

并不是所有的操作都由 handler 完成。例如，当 MySQL 需要进行表锁的时候。handler 可能会实现自己的级别的、更细粒度的锁，如 InnoDB 就实现了自己的行基本锁，但这并不能代替服务器层的表锁。正如我们第 1 章所介绍的，如果是所有存储引擎共有的特性则由服务器层实现，比如时间和日期函数、视图、触发器等。

为了执行查询，MySQL 只需要重复执行计划中的各个操作，直到完成所有的数据查询。

6.4.5 返回结果给客户端

查询执行的最后一个阶段是将结果返回给客户端。即使查询不需要返回结果集给客户端，MySQL 仍然会返回这个查询的一些信息，如该查询影响到的行数。

如果查询可以被缓存，那么 MySQL 在这个阶段也会将结果存放到查询缓存中。

MySQL 将结果集返回客户端是一个增量、逐步返回的过程。例如，我们回头看看前面的关联操作，一旦服务器处理完最后一个关联表，开始生成第一条结果时，MySQL 就可以开始向客户端逐步返回结果集了。

这样处理有两个好处：服务器端无须存储太多的结果，也就不会因为要返回太多结果而 229
消耗太多内存。另外，这样的处理也让 MySQL 客户端第一时间获得返回的结果[注22]。

结果集中的每一行都会以一个满足 MySQL 客户端 / 服务器通信协议的封包发送，再通过 TCP 协议进行传输，在 TCP 传输的过程中，可能对 MySQL 的封包进行缓存然后批量传输。

6.5 MySQL 查询优化器的局限性

MySQL 的万能“嵌套循环”并不是对每种查询都是最优的。不过还好，MySQL 查询优化器只对少部分查询不适用，而且我们往往可以通过改写查询让 MySQL 高效地完成工作。还有一个好消息，MySQL 5.6 版本正式发布后，会消除很多 MySQL 原本的限制，让更多的查询能够以尽可能高的效率完成。

6.5.1 关联子查询

MySQL 的子查询实现得非常糟糕。最糟糕的一类查询是 `WHERE` 条件中包含 `IN()` 的子查询语句。例如，我们希望找到 Sakila 数据库中，演员 Penelope Guiness（他的 `actor_id` 为 1）参演过的所有影片信息。很自然的，我们会按照下面的方式用子查询实现：

注 22： 可以通过一些办法来影响这个行为——例如，我们可以使用 SQL_BUFFER_RESULT。参考后面的“查询优化提示”。

```
mysql> SELECT * FROM sakila.film
    -> WHERE film_id IN(
    ->    SELECT film_id FROM sakila.film_actor WHERE actor_id = 1);
```

因为 MySQL 对 IN() 列表中的选项有专门的优化策略，一般会认为 MySQL 会先执行子查询返回所有包含 actor_id 为 1 的 film_id。一般来说，IN() 列表查询速度很快，所以我们会认为上面的查询会这样执行：

```
-- SELECT GROUP_CONCAT(film_id) FROM sakila.film_actor WHERE actor_id = 1;
-- Result: 1,23,25,106,140,166,277,361,438,499,506,509,605,635,749,832,939,970,980
SELECT * FROM sakila.film
WHERE film_id
IN(1,23,25,106,140,166,277,361,438,499,506,509,605,635,749,832,939,970,980);
```

很不幸，MySQL 不是这样做的。MySQL 会将相关的外层表压到子查询中，它认为这样可以更高效率地查找到数据行。也就是说，MySQL 会将查询改写成下面的样子：

```
SELECT * FROM sakila.film
WHERE EXISTS (
   SELECT * FROM sakila.film_actor WHERE actor_id = 1
   AND film_actor.film_id = film.film_id);
```

230 这时，子查询需要根据 film_id 来关联外部表 film，因为需要 film_id 字段，所以 MySQL 认为无法先执行这个子查询。通过 EXPLAIN 我们可以看到子查询是一个相关子查询（DEPENDENT SUBQUERY）（可以使用 EXPLAIN EXTENDED 来查看这个查询被改写成了什么样子）：

```
mysql> EXPLAIN SELECT * FROM sakila.film ...;
+----+--------------------+------------+--------+------------------------+
| id | select_type        | table      | type   | possible_keys          |
+----+--------------------+------------+--------+------------------------+
|  1 | PRIMARY            | film       | ALL    | NULL                   |
|  2 | DEPENDENT SUBQUERY | film_actor | eq_ref | PRIMARY,idx_fk_film_id |
+----+--------------------+------------+--------+------------------------+
```

根据 EXPLAIN 的输出我们可以看到，MySQL 先选择对 file 表进行全表扫描，然后根据返回的 film_id 逐个执行子查询。如果是一个很小的表，这个查询糟糕的性能可能还不会引起注意，但是如果外层的表是一个非常大的表，那么这个查询的性能会非常糟糕。当然我们很容易用下面的办法来重写这个查询：

```
mysql> SELECT film.* FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    -> WHERE actor_id = 1;
```

另一个优化的办法是使用函数 GROUP_CONCAT() 在 IN() 中构造一个由逗号分隔的列表。有时这比上面的使用关联改写更快。因为使用 IN() 加子查询，性能经常会非常糟，所以通常建议使用 EXISTS() 等效的改写查询来获取更好的效率。下面是另一种改写 IN() 加

子查询的办法：

```
mysql> SELECT * FROM sakila.film
    -> WHERE EXISTS(
    ->    SELECT * FROM sakila.film_actor WHERE actor_id = 1
    ->       AND film_actor.film_id = film.film_id);
```

> 这里讨论的优化器的限制直到 Oracle 推出的 MySQL 5.5 都一直存在。MySQL 的另一个分支 MariaDB 则在原有的优化器的基础上做了大量的改进，例如这里提到的 IN() 加子查询改进。

如何用好关联子查询

并不是所有关联子查询的性能都会很差。如果有人跟你说："别用关联子查询"，那么不要理他。先测试，然后做出自己的判断。很多时候，关联子查询是一种非常合理、自然，甚至是性能最好的写法。我们看看下面的例子：

```
mysql> EXPLAIN SELECT film_id, language_id FROM sakila.film
    -> WHERE NOT EXISTS(
    ->    SELECT * FROM sakila.film_actor
    ->    WHERE film_actor.film_id = film.film_id
    -> )\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 951
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: film_actor
         type: ref
possible_keys: idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: film.film_id
         rows: 2
        Extra: Using where; Using index
```

231

一般会建议使用左外连接（LEFT OUTER JOIN）重写该查询，以代替子查询。理论上，改写后 MySQL 的执行计划完全不会改变。我们来看这个例子：

```
mysql> EXPLAIN SELECT film.film_id, film.language_id
    -> FROM sakila.film
    ->    LEFT OUTER JOIN sakila.film_actor USING(film_id)
    -> WHERE film_actor.film_id IS NULL\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 951
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: sakila.film.film_id
         rows: 2
        Extra: Using where; Using index; Not exists
```

232 可以看到，这里的执行计划基本上一样，下面是一些微小的区别：

- 表 film_actor 的访问类型一个是 DEPENDENT SUBQUERY，而另一个是 SIMPLE。这个不同是由于语句的写法不同导致的，一个是普通查询，一个是子查询。这对底层存储引擎接口来说，没有任何不同。
- 对 film 表，第二个查询的 Extra 中没有“Using where”，但这不重要，第二个查询的 USING 子句和第一个查询的 WHERE 子句实际上是完全一样的。
- 在第二个表 film_actor 的执行计划的 Extra 列有“Not exists”。这是我们前面章节中提到的提前终止算法（early-termination algorithm），MySQL 通过使用“Not exists”优化来避免在表 film_actor 的索引中读取任何额外的行。这完全等效于直接编写 NOT EXISTS 子查询，这个执行计划中也是一样，一旦匹配到一行数据，就立刻停止扫描。

所以，从理论上讲，MySQL 将使用完全相同的执行计划来完成这个查询。现实世界中，我们建议通过一些测试来判断使用哪种写法速度会更快。针对上面的案例，我们对两种写法进行了测试，表 6-1 中列出了测试结果。

表6-1：NOT EXISTS 和左外连接的性能比较

查询	每秒查询数结果（QPS）
NOT EXISTS 子查询	360 QPS
LEFT OUTER JOIN	425 QPS

我们的测试显示，使用子查询的写法要略微慢些！

不过每个具体的案例会各有不同，有时候子查询写法也会快些。例如，当返回结果中只有一个表中的某些列的时候。听起来，这种情况对于关联查询效率也会很好。具体情况具体分析，例如下面的关联，我们希望返回所有包含同一个演员参演的电影，因为一个电影会有很多演员参演，所以可能会返回一些重复的记录：

```
mysql> SELECT film.film_id FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id);
```

我们需要使用 DISTINCT 和 GROUP BY 来移除重复的记录：

```
mysql> SELECT DISTINCT film.film_id FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id);
```

但是，回头看看这个查询，到底这个查询返回的结果集意义是什么？至少这样的写法会让 SQL 的意义很不明显。如果使用 EXISTS 则很容易表达"包含同一个参演演员"的逻辑，而且不需要使用 DISTINCT 和 GROUP BY，也不会产生重复的结果集，我们知道一旦使用了 DISTINCT 和 GROUP BY，那么在查询的执行过程中，通常需要产生临时中间表。下面我们用子查询的写法替换上面的关联：

233

```
mysql> SELECT film_id FROM sakila.film
    ->    WHERE EXISTS(SELECT * FROM sakila.film_actor
    ->    WHERE film.film_id = film_actor.film_id);
```

再一次，我们需要通过测试来对比这两种写法，哪个更快一些。测试结果参考表 6-2。

表6-2：EXISTS和关联性能对比

查询	每秒查询数结果（QPS）
INNER JOIN	185 QPS
EXISTS 子查询	325 QPS

在这个案例中，我们看到子查询速度要比关联查询更快些。

通过上面这个详细的案例，主要想说明两点：一是不需要听取那些关于子查询的"绝对真理"，二是应该用测试来验证对子查询的执行计划和响应时间的假设。最后，关于子查询我们需要提到的是一个 MySQL 的 bug。在 MYSQL 5.1.48 和之前的版本中，下面的写法会锁住 table2 中的一条记录：

```
SELECT ... FROM table1 WHERE col = (SELECT ... FROM table2 WHERE ...);
```

如果遇到该 bug，子查询在高并发情况下的性能，就会和在单线程测试时的性能相差甚远。

这个 bug 的编号是 46947，虽然这个问题已经被修复了，但是我们仍然要提醒读者：不要主观猜测，应该通过测试来验证猜想。

6.5.2 UNION 的限制

有时，MySQL 无法将限制条件从外层“下推”到内层，这使得原本能够限制部分返回结果的条件无法应用到内层查询的优化上。

如果希望 UNION 的各个子句能够根据 LIMIT 只取部分结果集，或者希望能够先排好序再合并结果集的话，就需要在 UNION 的各个子句中分别使用这些子句。例如，想将两个子查询结果联合起来，然后再取前 20 条记录，那么 MySQL 会将两个表都存放到同一个临时表中，然后再取出前 20 行记录：

```
(SELECT first_name, last_name
 FROM sakila.actor
 ORDER BY last_name)
UNION ALL
(SELECT first_name, last_name
 FROM sakila.customer
 ORDER BY last_name)
LIMIT 20;
```

这条查询将会把 actor 中的 200 条记录和 customer 表中的 599 条记录存放在一个临时表中，然后再从临时表中取出前 20 条。可以通过在 UNION 的两个子查询中分别加上一个 LIMIT 20 来减少临时表中的数据：

234

```
(SELECT first_name, last_name
 FROM sakila.actor
 ORDER BY last_name
 LIMIT 20)
UNION ALL
(SELECT first_name, last_name
 FROM sakila.customer
 ORDER BY last_name
 LIMIT 20)
LIMIT 20;
```

现在中间的临时表只会包含 40 条记录了，除了性能考虑之外，在这里还需要注意一点：从临时表中取出数据的顺序并不是一定的，所以如果想获得正确的顺序，还需要加上一个全局的 ORDER BY 和 LIMIT 操作。

6.5.3 索引合并优化

在前面的章节已经讨论过，在 5.0 和更新的版本中，当 WHERE 子句中包含多个复杂条件的时候，MySQL 能够访问单个表的多个索引以合并和交叉过滤的方式来定位需要查找的行。

6.5.4 等值传递

某些时候，等值传递会带来一些意想不到的额外消耗。例如，有一个非常大的 IN() 列表，而 MySQL 优化器发现存在 WHERE、ON 或者 USING 的子句，将这个列表的值和另一个表的某个列相关联。

那么优化器会将 IN() 列表都复制应用到关联的各个表中。通常，因为各个表新增了过滤条件，优化器可以更高效地从存储引擎过滤记录。但是如果这个列表非常大，则会导致优化和执行都会变慢。在本书写作的时候，除了修改 MySQL 源代码，目前还没有什么办法能够绕过该问题（不过这个问题很少会碰到）。

6.5.5 并行执行

MySQL 无法利用多核特性来并行执行查询。很多其他的关系型数据库能够提供这个特性，但是 MySQL 做不到。这里特别指出是想告诉读者不要花时间去尝试寻找并行执行查询的方法。

6.5.6 哈希关联

在本书写作的时候，MySQL 并不支持哈希关联——MySQL 的所有关联都是嵌套循环关联。不过，可以通过建立一个哈希索引来曲线地实现哈希关联。如果使用的是 Memory 存储引擎，则索引都是哈希索引，所以关联的时候也类似于哈希关联。可以参考第 5 章的“创建自定义哈希索引”部分。另外，MariaDB 已经实现了真正的哈希关联。 235

6.5.7 松散索引扫描[注 23]

由于历史原因，MySQL 并不支持松散索引扫描，也就无法按照不连续的方式扫描一个索引。通常，MySQL 的索引扫描需要先定义一个起点和终点，即使需要的数据只是这段索引中很少数的几个，MySQL 仍需要扫描这段索引中每一个条目。

下面我们通过一个示例说明这点。假设我们有如下索引 (a, b)，有下面的查询：

```
mysql> SELECT ... FROM tbl WHERE b BETWEEN 2 AND 3;
```

因为索引的前导字段是列 a，但是在查询中只指定了字段 b，MySQL 无法使用这个索引，从而只能通过全表扫描找到匹配的行，如图 6-5 所示。

注 23： 相当于 Oracle 中的跳跃索引扫描（skip index scan）。——译者注

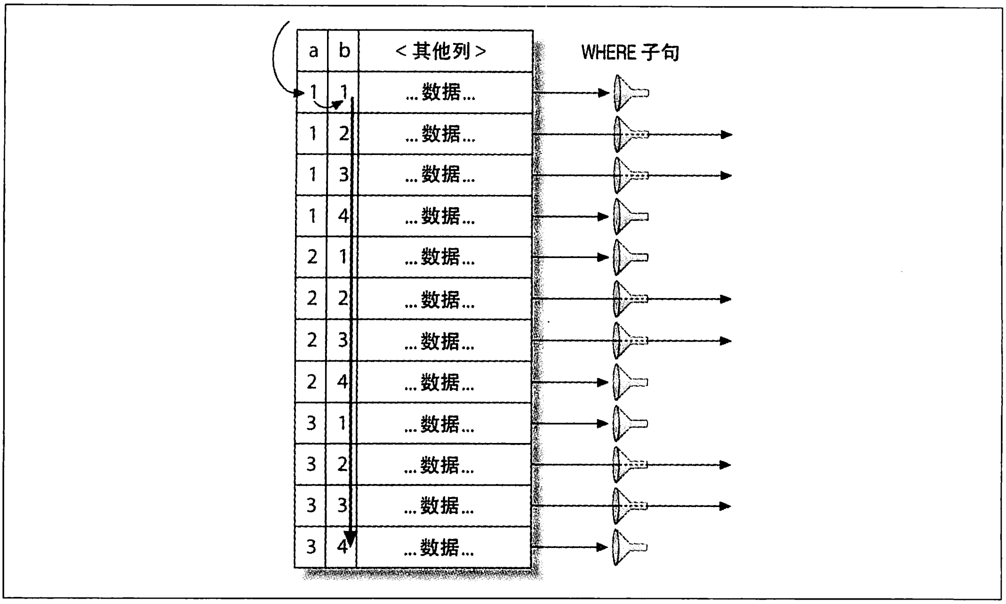


图6-5：MySQL通过全表扫描找到需要的记录

了解索引的物理结构的话，不难发现还可以有一个更快的办法执行上面的查询。索引的物理结构（不是存储引擎的 API）使得可以先扫描 a 列第一个值对应的 b 列的范围，然后再跳到 a 列第二个不同值扫描对应的 b 列的范围。图 6-6 展示了如果由 MySQL 来实现这个过程会怎样。

236

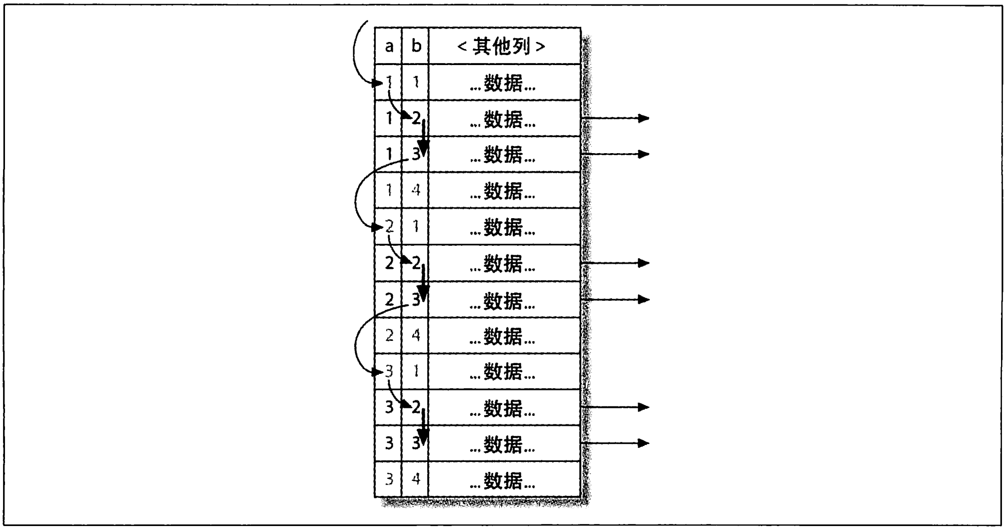


图6-6：使用松散索引扫描效率会更高，但是MySQL现在还不支持这么做

注意到，这时就无须再使用 WHERE 子句过滤，因为松散索引扫描已经跳过了所有不需要的记录。

上面是一个简单的例子，除了松散索引扫描，新增一个合适的索引当然也可以优化上述查询。但对于某些场景，增加索引是没用的，例如，对于第一个索引列是范围条件，第二个索引列是等值条件的查询，靠增加索引就无法解决问题。

MySQL 5.0 之后的版本，在某些特殊的场景下是可以使用松散索引扫描的，例如，在一个分组查询中需要找到分组的最大值和最小值：

```
mysql> EXPLAIN SELECT actor_id, MAX(film_id)
    -> FROM sakila.film_actor
    -> GROUP BY actor_id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: range
possible_keys: NULL
          key: PRIMARY
      key_len: 2
          ref: NULL
         rows: 396
        Extra: Using index for group-by
```

在 EXPLAIN 中的 Extra 字段显示“Using index for group-by”，表示这里将使用松散索引扫描，不过如果 MySQL 能写上“loose index probe”，相信会更好理解。

在 MySQL 很好地支持松散索引扫描之前，一个简单的绕过问题的办法就是给前面的列 237
加上可能的常数值。在前面索引案例学习的章节中，我们已经看到这样做的好处了。

在 MySQL 5.6 之后的版本，关于松散索引扫描的一些限制将会通过“索引条件下推（index condition pushdown）”的方式解决。

6.5.8 最大值和最小值优化

对于 MIN() 和 MAX() 查询，MySQL 的优化做得并不好。这里有一个例子：

```
mysql> SELECT MIN(actor_id) FROM sakila.actor WHERE first_name = 'PENELOPE';
```

因为在 first_name 字段上并没有索引，因此 MySQL 将会进行一次全表扫描。如果 MySQL 能够进行主键扫描，那么理论上，当 MySQL 读到第一个满足条件的记录的时候，就是我们需要找的最小值了，因为主键是严格按照 actor_id 字段的大小顺序排列的。但是 MySQL 这时只会做全表扫描，我们可以通过查看 SHOW STATUS 的全表扫描计数器来验证这一点。一个曲线的优化办法是移除 MIN()，然后使用 LIMIT 来将查询重写如下：

```
mysql> SELECT actor_id FROM sakila.actor USE INDEX(PRIMARY)
    -> WHERE first_name = 'PENELOPE' LIMIT 1;
```

这个策略可以让 MySQL 扫描尽可能少的记录数。如果你是一个完美主义者，可能会说这个 SQL 已经无法表达她的本意了。一般我们通过 SQL 告诉服务器我们需要什么数据，由服务器来决定如何最优地获取数据，不过在这个案例中，我们其实是告诉 MySQL 如何去获取我们需要的数据，通过 SQL 并不能一眼就看出我们其实是想要一个最小值。确实如此，有时候为了获得更高的性能，我们不得不放弃一些原则。

6.5.9 在同一个表上查询和更新

MySQL 不允许对同一张表同时进行查询和更新。这其实并不是优化器的限制，如果清楚 MySQL 是如何执行查询的，就可以避免这种情况。下面是一个无法运行的 SQL，虽然这是一个符合标准的 SQL 语句。这个 SQL 语句尝试将两个表中相似行的数量记录到字段 cnt 中：

```
mysql> UPDATE tbl AS outer_tbl
    ->    SET cnt = (
    ->       SELECT count(*) FROM tbl AS inner_tbl
    ->       WHERE inner_tbl.type = outer_tbl.type
    ->    );
ERROR 1093 (HY000): You can't specify target table 'outer_tbl' for update in FROM
clause
```

238 可以通过使用生成表的形式来绕过上面的限制，因为 MySQL 只会把这个表当作一个临时表来处理。实际上，这执行了两个查询：一个是子查询中的 SELECT 语句，另一个是多表关联 UPDATE，只是关联的表是一个临时表。子查询会在 UPDATE 语句打开表之前就完成，所以下面的查询将会正常执行：

```
mysql> UPDATE tbl
    ->    INNER JOIN(
    ->       SELECT type, count(*) AS cnt
    ->       FROM tbl
    ->       GROUP BY type
    ->    ) AS der USING(type)
    -> SET tbl.cnt = der.cnt;
```

6.6 查询优化器的提示（hint）

如果对优化器选择的执行计划不满意，可以使用优化器提供的几个提示（hint）来控制最终的执行计划。下面将列举一些常见的提示，并简单地给出什么时候使用该提示。通过在查询中加入相应的提示，就可以控制该查询的执行计划。关于每个提示的具体用法，

建议直接阅读 MySQL 官方手册。有些提示和版本有直接关系。可以使用的一些提示如下：

HIGH_PRIORITY 和 LOW_PRIORITY

这个提示告诉 MySQL，当多个语句同时访问某一个表的时候，哪些语句的优先级相对高些、哪些语句的优先级相对低些。

HIGH_PRIORITY 用于 SELECT 语句的时候，MySQL 会将此 SELECT 语句重新调度到所有正在等待表锁以便修改数据的语句之前。实际上 MySQL 是将其放在表的队列的最前面，而不是按照常规顺序等待。HIGH_PRIORITY 还可以用于 INSERT 语句，其效果只是简单地抵消了全局 LOW_PRIORITY 设置对该语句的影响。

LOW_PRIORITY 则正好相反：它会让该语句一直处于等待状态，只要队列中还有需要访问同一个表的语句——即使是那些比该语句还晚提交到服务器的语句。这就像一个过于礼貌的人站在餐厅门口，只要还有其他顾客在等待就一直不进去，很明显这容易把自己给饿坏。LOW_PRIORITY 提示在 SELECT、INSERT、UPDATE 和 DELETE 语句中都可以使用。

这两个提示只对使用表锁的存储引擎有效，千万不要在 InnoDB 或者其他有细粒度锁机制和并发控制的引擎中使用。即使是在 MyISAM 中使用也要注意，因为这两个提示会导致并发插入被禁用，可能会严重降低性能。

HIGH_PRIORITY 和 LOW_PRIORITY 经常让人感到困惑。这两个提示并不会获取更多资源让查询"积极"工作，也不会少获取资源让查询"消极"工作。它们只是简单地控制了 MySQL 访问某个数据表的队列顺序。

DELAYED 239

这个提示对 INSERT 和 REPLACE 有效。MySQL 会将使用该提示的语句立即返回给客户端，并将插入的行数据放入到缓冲区，然后在表空闲时批量将数据写入。日志系统使用这样的提示非常有效，或者是其他需要写入大量数据但是客户端却不需要等待单条语句完成 I/O 的应用。这个用法有一些限制：并不是所有的存储引擎都支持这样的做法；并且该提示会导致函数 LAST_INSERT_ID() 无法正常工作。

STRAIGHT_JOIN

这个提示可以放置在 SELECT 语句的 SELECT 关键字之后，也可以放置在任何两个关联表的名字之间。第一个用法是让查询中所有的表按照在语句中出现的顺序进行关联。第二个用法则是固定其前后两个表的关联顺序。

当 MySQL 没能选择正确的关联顺序的时候，或者由于可能的顺序太多导致 MySQL 无法评估所有的关联顺序的时候，STRAIGHT_JOIN 都会很有用。在后面这种情况，MySQL 可能会花费大量时间在"statistics"状态，加上这个提示则会大大减少优化器的搜索空间。

可以先使用 EXPLAIN 语句来查看优化器选择的关联顺序，然后使用该提示来重写查询，再看看它的关联顺序。当你确定无论怎样的 where 条件，某个固定的关联顺序始终是最佳的时候，使用这个提示可以大大提高优化器的效率。但是在升级 MySQL 版本的时候，需要重新审视下这类查询，某些新的优化特性可能会因为该提示而失效。

SQL_SMALL_RESULT 和 SQL_BIG_RESULT

这两个提示只对 SELECT 语句有效。它们告诉优化器对 GROUP BY 或者 DISTINCT 查询如何使用临时表及排序。SQL_SMALL_RESULT 告诉优化器结果集会很小，可以将结果集放在内存中的索引临时表，以避免排序操作。如果是 SQL_BIG_RESULT，则告诉优化器结果集可能会非常大，建议使用磁盘临时表做排序操作。

SQL_BUFFER_RESULT

这个提示告诉优化器将查询结果放入到一个临时表，然后尽可能快地释放表锁。这和前面提到的由客户端缓存结果不同。当你没法使用客户端缓存的时候，使用服务器端的缓存通常很有效。带来的好处是无须在客户端上消耗太多的内存，还可以尽可能快地释放对应的表锁。代价是，服务器端将需要更多的内存。

SQL_CACHE 和 SQL_NO_CACHE

这个提示告诉 MySQL 这个结果集是否应该缓存在查询缓存中，下一章我们将详细介绍如何使用。

SQL_CALC_FOUND_ROWS

严格来说，这并不是一个优化器提示。它不会告诉优化器任何关于执行计划的东西。它会让 MySQL 返回的结果集包含更多的信息。查询中加上该提示 MySQL 会计算
240 除去 LIMIT 子句后这个查询要返回的结果集的总数，而实际上只返回 LIMIT 要求的结果集。可以通过函数 FOUND_ROW() 获得这个值。（参阅后面的“SQL_CALC_FOUND_ROWS 优化”部分，了解下为什么不应该使用该提示。）

FOR UPDATE 和 LOCK IN SHARE MODE

这也不是真正的优化器提示。这两个提示主要控制 SELECT 语句的锁机制，但只对实现了行级锁的存储引擎有效。使用该提示会对符合查询条件的数据行加锁。对于 INSERT...SELECT 语句是不需要这两个提示的，因为对于 MySQL 5.0 和更新版本会默认给这些记录加上读锁。（可以禁用该默认行为，但不是个好主意，在后面关于复制和备份的章节中将解释这一点。）

唯一内置的支持这两个提示的引擎就是 InnoDB。另外需要记住的是，这两个提示会让某些优化无法正常使用，例如索引覆盖扫描。InnoDB 不能在不访问主键的情况下排他地锁定行，因为行的版本信息保存在主键中。

糟糕的是，这两个提示经常被滥用，很容易造成服务器的锁争用问题，后面章节我

们将讨论这点。应该尽可能地避免使用这两个提示，通常都有其他更好的方式可以实现同样的目的。

USE INDEX、IGNORE INDEX 和 FORCE INDEX

这几个提示会告诉优化器使用或者不使用哪些索引来查询记录（例如，在决定关联顺序的时候使用哪个索引）。在 MySQL 5.0 和更早的版本，这些提示并不会影响到优化器选择哪个索引进行排序和分组，在 MyQL 5.1 和之后的版本可以通过新增选项 FOR ORDER BY 和 FOR GROUP BY 来指定是否对排序和分组有效。

FORCE INDEX 和 USE INDEX 基本相同，除了一点：FORCE INDEX 会告诉优化器全表扫描的成本会远远高于索引扫描，哪怕实际上该索引用处不大。当发现优化器选择了错误的索引，或者因为某些原因（比如在不使用 ORDER BY 的时候希望结果有序）要使用另一个索引时，可以使用该提示。在前面关于如何使用 LIMIT 高效地获取最小值的案例中，已经演示过这种用法。

在 MySQL 5.0 和更新版本中，新增了一些参数用来控制优化器的行为：

optimizer_search_depth

这个参数控制优化器在穷举执行计划时的限度。如果查询长时间处于“Statistics”状态，那么可以考虑调低此参数。

optimizer_prune_level

该参数默认是打开的，这让优化器会根据需要扫描的行数来决定是否跳过某些执行计划。

optimizer_switch 241

这个变量包含了一些开启 / 关闭优化器特性的标志位。例如在 MySQL 5.1 中可以通过这个参数来控制禁用索引合并的特性。

前两个参数是用来控制优化器可以走的一些“捷径”。这些捷径可以让优化器在处理非常复杂的 SQL 语句时，仍然可以很高效，但这也可能让优化器错过一些真正最优的执行计划。所以应该根据实际需要来修改这些参数。

MySQL 升级后的验证

在优化器面前耍一些“小聪明”是不好的。这样做收效甚小，但是却给维护带来了很多额外的工作量。在 MySQL 版本升级的时候，这个问题就很突出了，你设置的“优化器提示”很可能会让新版的优化策略失效。

MySQL 5.0 版本引入了大量优化策略，在还没有正式发布的 5.6 版本中，优化器的改进也是近些年来最大的一次改进。如果要更新到这些版本，当然希望能够从这些改进中受益。

新版 MySQL 基本上在各个方面都有非常大的改进，5.5 和 5.6 这两个版本尤为突出。升级操作一般来说都很顺利，但仍然建议仔细检查各个细节，以防止一些边界情况影响你的应用程序。不过还好，要避免这些，你不需要付出太多的精力。使用 Percona Toolkit 中的 pt-upgrade 工具，就可以检查在新版本中运行的 SQL 是否与老版本一样，返回相同的结果。

6.7 优化特定类型的查询

这一节，我们将介绍如何优化特定类型的查询。在本书的其他部分都会分散介绍这些优化技巧，不过这里将会汇总一下，以便参考和查阅。

本节介绍的多数优化技巧都是和特定的版本有关的，所以对于未来 MySQL 的版本未必适用。毫无疑问，某一天优化器自己也会实现这里列出的部分或者全部优化技巧。

6.7.1 优化 COUNT() 查询

COUNT() 聚合函数，以及如何优化使用了该函数的查询，很可能是 MySQL 中最容易被误解的前 10 个话题之一。在网上随便搜索一下就能看到很多错误的理解，可能比我们想象的多得多。

在做优化之前，先来看看 COUNT() 函数真正的作用是什么。

242

COUNT() 的作用

COUNT() 是一个特殊的函数，有两种非常不同的作用：它可以统计某个列值的数量，也可以统计行数。在统计列值时要求列值是非空的（不统计 NULL）。如果在 COUNT() 的括号中指定了列或者列的表达式，则统计的就是这个表达式有值的结果数[注24]。因为很多人对 NULL 理解有问题，所以这里很容易产生误解。如果想了解更多关于 SQL 语句中 NULL 的含义，建议阅读一些关于 SQL 语句基础的书籍。（关于这个话题，互联网上的一些信息是不够精确的。）

COUNT() 的另一个作用是统计结果集的行数。当 MySQL 确认括号内的表达式值不可能

注 24： 而不是 NULL。——译者注

为空时，实际上就是在统计行数。最简单的就是当我们使用 COUNT(*) 的时候，这种情况下通配符 * 并不会像我们猜想的那样扩展成所有的列，实际上，它会忽略所有的列而直接统计所有的行数。

我们发现一个最常见的错误就是，在括号内指定了一个列却希望统计结果集的行数。如果希望知道的是结果集的行数，最好使用 COUNT(*)，这样写意义清晰，性能也会很好。

关于 MyISAM 的神话

一个容易产生的误解就是：MyISAM 的 COUNT() 函数总是非常快，不过这是有前提条件的，即只有没有任何 WHERE 条件的 COUNT(*) 才非常快，因为此时无须实际地去计算表的行数。MySQL 可以利用存储引擎的特性直接获得这个值。如果 MySQL 知道某列 col 不可能为 NULL 值，那么 MySQL 内部会将 COUNT(col) 表达式优化为 COUNT(*)。

当统计带 WHERE 子句的结果集行数，可以是统计某个列值的数量时，MyISAM 的 COUNT() 和其他存储引擎没有任何不同，就不再有神话般的速度了。所以在 MyISAM 引擎表上执行 COUNT() 有时候比别的引擎快，有时候比别的引擎慢，这受很多因素影响，要视具体情况而定。

简单的优化

有时候可以使用 MyISAM 在 COUNT(*) 全表非常快的这个特性，来加速一些特定条件的 COUNT() 的查询。在下面的例子中，我们使用标准数据库 world 来看看如何快速查找到所有 ID 大于 5 的城市。可以像下面这样来写这个查询：

```
mysql> SELECT COUNT(*) FROM world.City WHERE ID > 5;
```

通过 SHOW STATUS 的结果可以看到该查询需要扫描 4 097 行数据。如果将条件反转一下，先查找 ID 小于等于 5 的城市数，然后用总城市数一减就能得到同样的结果，却可以将扫描的行数减少到 5 行以内：

```
mysql> SELECT (SELECT COUNT(*) FROM world.City) - COUNT(*)
    -> FROM world.City WHERE ID <= 5;
```

243

这样做可以大大减少需要扫描的行数，是因为在查询优化阶段会将其中的子查询直接当作一个常数来处理，我们可以通过 EXPLAIN 来验证这点：

```
+----+-------------+-------+...+------+------------------------------+
| id | select_type | table |...| rows | Extra                        |
+----+-------------+-------+...+------+------------------------------+
|  1 | PRIMARY     | City  |...|    6 | Using where; Using index     |
|  2 | SUBQUERY    | NULL  |...| NULL | Select tables optimized away |
+----+-------------+-------+...+------+------------------------------+
```

在邮件组和IRC聊天频道中，通常会看到这样的问题：如何在同一个查询中统计同一个列的不同值的数量，以减少查询的语句量。例如，假设可能需要通过一个查询返回各种不同颜色的商品数量，此时不能使用OR语句（比如SELECT COUNT(color='blue' OR color='red') FROM items;），因为这样做就无法区分不同颜色的商品数量；也不能在WHERE条件中指定颜色（比如SELECT COUNT(*) FROM items WHERE color='blue' AND color='RED';），因为颜色的条件是互斥的。下面的查询可以在一定程度上解决这个问题[注25]。

```
mysql> SELECT SUM(IF(color = 'blue', 1, 0)) AS blue,SUM(IF(color = 'red', 1, 0))
    -> AS red FROM items;
```

也可以使用COUNT()而不是SUM()实现同样的目的，只需要将满足条件设置为真，不满足条件设置为NULL即可：

```
mysql> SELECT COUNT(color = 'blue' OR NULL) AS blue, COUNT(color = 'red' OR NULL)
    -> AS red FROM items;
```

使用近似值

有时候某些业务场景并不要求完全精确的COUNT值，此时可以用近似值来代替。EXPLAIN出来的优化器估算的行数就是一个不错的近似值，执行EXPLAIN并不需要真正地去执行查询，所以成本很低。

很多时候，计算精确值的成本非常高，而计算近似值则非常简单。曾经有一个客户希望我们统计他的网站的当前活跃用户数是多少，这个活跃用户数保存在缓存中，过期时间为30分钟，所以每隔30分钟需要重新计算并放入缓存。因此这个活跃用户数本身就不是精确值，所以使用近似值代替是可以接受的。另外，如果要精确统计在线人数，通常WHERE条件会很复杂，一方面需要剔除当前非活跃用户，另一方面还要剔除系统中某些特定ID的“默认”用户，去掉这些约束条件对总数的影响很小，但却可能很好地提升该查询的性能。更进一步地优化则可以尝试删除DISTINCT这样的约束来避免文件排序。这样重写过的查询要比原来的精确统计的查询快很多，而返回的结果则几乎相同。

244

更复杂的优化

通常来说，COUNT()都需要扫描大量的行（意味着要访问大量数据）才能获得精确的结果，因此是很难优化的。除了前面的方法，在MySQL层面还能做的就只有索引覆盖扫描了。如果这还不够，就需要考虑修改应用的架构，可以增加汇总表（第4章已经介绍过），或者增加类似*Memcached*这样的外部缓存系统。可能很快你就会发现陷入到一个熟悉的困境，“快速，精确和实现简单”，三者永远只能满足其二，必须舍掉其中一个。

注25：也可以写成这样的SUM()表达式：SUM(color = 'blue')，SUM(color = 'red')。

6.7.2 优化关联查询

这个话题基本上整本书都在讨论，这里需要特别提到的是：

- 确保 ON 或者 USING 子句中的列上有索引。在创建索引的时候就要考虑到关联的顺序。当表 A 和表 B 用列 c 关联的时候，如果优化器的关联顺序是 B、A，那么就不需要在 B 表的对应列上建上索引。没有用到的索引只会带来额外的负担。一般来说，除非有其他理由，否则只需要在关联顺序中的第二个表的相应列上创建索引。
- 确保任何的 GROUP BY 和 ORDER BY 中的表达式只涉及到一个表中的列，这样 MySQL 才有可能使用索引来优化这个过程。
- 当升级 MySQL 的时候需要注意：关联语法、运算符优先级等其他可能会发生变化的地方。因为以前是普通关联的地方可能会变成笛卡儿积，不同类型的关联可能会生成不同的结果等。

6.7.3 优化子查询

关于子查询优化我们给出的最重要的优化建议就是尽可能使用关联查询代替，至少当前的 MySQL 版本需要这样。本章的前面章节已经详细介绍了这点。"尽可能使用关联"并不是绝对的，如果使用的是 MySQL 5.6 或更新的版本或者 MariaDB，那么就可以直接忽略关于子查询的这些建议了。

6.7.4 优化 GROUP BY 和 DISTINCT

在很多场景下，MySQL 都使用同样的办法优化这两种查询，事实上，MySQL 优化器会在内部处理的时候相互转化这两类查询。它们都可以使用索引来优化，这也是最有效的优化办法。

在 MySQL 中，当无法使用索引的时候，GROUP BY 使用两种策略来完成：使用临时表或者文件排序来做分组。对于任何查询语句，这两种策略的性能都有可以提升的地方。可以通过使用提示 SQL_BIG_RESULT 和 SQL_SMALL_RESULT 来让优化器按照你希望的方式运行。在本章的前面章节我们已经讨论了这点。 245

如果需要对关联查询做分组（GROUP BY），并且是按照查找表中的某个列进行分组，那么通常采用查找表的标识列分组的效率会比其他列更高。例如下面的查询效率不会很好：

```
mysql> SELECT actor.first_name, actor.last_name, COUNT(*)
    -> FROM sakila.film_actor
    ->    INNER JOIN sakila.actor USING(actor_id)
    -> GROUP BY actor.first_name, actor.last_name;
```

如果查询按照下面的写法效率则会更高：

```
mysql> SELECT actor.first_name, actor.last_name, COUNT(*)
    -> FROM sakila.film_actor
    ->    INNER JOIN sakila.actor USING(actor_id)
    -> GROUP BY film_actor.actor_id;
```

使用 actor.actor_id 列分组的效率甚至会比使用 film_actor.actor_id 更好。这一点通过简单的测试即可验证。

这个查询利用了演员的姓名和 ID 直接相关的特点，因此改写后的结果不受影响，但显然不是所有的关联语句的分组查询都可以改写成在 SELECT 中直接使用非分组列的形式的。甚至可能会在服务器上设置 SQL_MODE 来禁止这样的写法。如果是这样，也可以通过 MIN() 或者 MAX() 函数来绕过这种限制，但一定要清楚，SELECT 后面出现的非分组列一定是直接依赖分组列，并且在每个组内的值是唯一的，或者是业务上根本不在乎这个值具体是什么：

```
mysql> SELECT MIN(actor.first_name), MAX(actor.last_name), ...;
```

较真的人可能会说这样写的分组查询是有问题的，确实如此。从 MIN() 或者 MAX() 函数的用法就可以看出这个查询是有问题的。但若更在乎的是 MySQL 运行查询的效率时这样做也无可厚非。如果实在较真的话也可以改写成下面的形式：

```
mysql> SELECT actor.first_name, actor.last_name, c.cnt
    -> FROM sakila.actor
    ->    INNER JOIN (
    ->       SELECT actor_id, COUNT(*) AS cnt
    ->       FROM sakila.film_actor
    ->       GROUP BY actor_id
    ->    ) AS c USING(actor_id) ;
```

这样写更满足关系理论，但成本有点高，因为子查询需要创建和填充临时表，而子查询中创建的临时表是没有任何索引的[注26]。

在分组查询的 SELECT 中直接使用非分组列通常都不是什么好主意，因为这样的结果通常是不定的，当索引改变，或者优化器选择不同的优化策略时都可能导致结果不一样。我

246 们碰到的大多数这种查询最后都导致了故障（因为 MySQL 不会对这类查询返回错误），而且这种写法大部分是由于偷懒而不是为优化而故意这么设计的。建议始终使用含义明确的语法。事实上，我们建议将 MySQL 的 SQL_MODE 设置为包含 ONLY_FULL_GROUP_BY，这时 MySQL 会对这类查询直接返回一个错误，提醒你需要重写这个查询。

如果没有通过 ORDER BY 子句显式地指定排序列，当查询使用 GROUP BY 子句的时候，结

注 26： 值得一提的是，MariaDB 修复了这个限制。

果集会自动按照分组的字段进行排序。如果不关心结果集的顺序，而这种默认排序又导致了需要文件排序，则可以使用 ORDER BY NULL，让 MySQL 不再进行文件排序。也可以在 GROUP BY 子句中直接使用 DESC 或者 ASC 关键字，使分组的结果集按需要的方向排序。

优化 GROUP BY WITH ROLLUP

分组查询的一个变种就是要求 MySQL 对返回的分组结果再做一次超级聚合。可以使用 WITH ROLLUP 子句来实现这种逻辑，但可能会不够优化。可以通过 EXPLAIN 来观察其执行计划，特别要注意分组是否是通过文件排序或者临时表实现的。然后再去掉 WITH ROLLUP 子句看执行计划是否相同。也可以通过本节前面介绍的优化器提示来固定执行计划。

很多时候，如果可以，在应用程序中做超级聚合是更好的，虽然这需要返回给客户端更多的结果。也可以在 FROM 子句中嵌套使用子查询，或者是通过一个临时表存放中间数据，然后和临时表执行 UNION 来得到最终结果。

最好的办法是尽可能的将 WITH ROLLUP 功能转移到应用程序中处理。

6.7.5 优化 LIMIT 分页

在系统中需要进行分页操作的时候，我们通常会使用 LIMIT 加上偏移量的办法实现，同时加上合适的 ORDER BY 子句。如果有对应的索引，通常效率会不错，否则，MySQL 需要做大量的文件排序操作。

一个非常常见又令人头疼的问题就是，在偏移量非常大的时候[注27]，例如可能是 LIMIT 1000,20 这样的查询，这时 MySQL 需要查询 10 020 条记录然后只返回最后 20 条，前面 10 000 条记录都将被抛弃，这样的代价非常高。如果所有的页面被访问的频率都相同，那么这样的查询平均需要访问半个表的数据。要优化这种查询，要么是在页面中限制分页的数量，要么是优化大偏移量的性能。

优化此类分页查询的一个最简单的办法就是尽可能地使用索引覆盖扫描，而不是查询所有的列。然后根据需要做一次关联操作再返回所需的列。对于偏移量很大的时候，这样做的效率会提升非常大。考虑下面的查询：

```
mysql> SELECT film_id, description FROM sakila.film ORDER BY title LIMIT 50, 5;
```

如果这个表非常大，那么这个查询最好改写成下面的样子： 247

注 27： 翻页到非常靠后的页面。——译者注

```
mysql> SELECT film.film_id, film.description
    -> FROM sakila.film
    ->    INNER JOIN (
    ->       SELECT film_id FROM sakila.film
    ->       ORDER BY title LIMIT 50, 5
    ->    ) AS lim USING(film_id);
```

这里的“延迟关联”将大大提升查询效率，它让 MySQL 扫描尽可能少的页面，获取需要访问的记录后再根据关联列回原表查询需要的所有列。这个技术也可以用于优化关联查询中的 LIMIT 子句。

有时候也可以将 LIMIT 查询转换为已知位置的查询，让 MySQL 通过范围扫描获得到对应的结果。例如，如果在一个位置列上有索引，并且预先计算出了边界值，上面的查询就可以改写为：

```
mysql> SELECT film_id, description FROM sakila.film
    -> WHERE position BETWEEN 50 AND 54 ORDER BY position;
```

对数据进行排名的问题也与此类似，但往往还会同时和 GROUP BY 混合使用。在这种情况下通常都需要预先计算并存储排名信息。

LIMIT 和 OFFSET 的问题，其实是 OFFSET 的问题，它会导致 MySQL 扫描大量不需要的行然后再抛弃掉。如果可以使用书签记录上次取数据的位置，那么下次就可以直接从该书签记录的位置开始扫描，这样就可以避免使用 OFFSET。例如，若需要按照租借记录做翻页，那么可以根据最新一条租借记录向后追溯，这种做法可行是因为租借记录的主键是单调增长的。首先使用下面的查询获得第一组结果：

```
mysql> SELECT * FROM sakila.rental
    -> ORDER BY rental_id DESC LIMIT 20;
```

假设上面的查询返回的是主键为 16 049 到 16 030 的租借记录，那么下一页查询就可以从 16 030 这个点开始：

```
mysql> SELECT * FROM sakila.rental
    -> WHERE rental_id < 16030
    -> ORDER BY rental_id DESC LIMIT 20;
```

该技术的好处是无论翻页到多么后面，其性能都会很好。

其他优化办法还包括使用预先计算的汇总表，或者关联到一个冗余表，冗余表只包含主键列和需要做排序的数据列。还可以使用 Sphinx 优化一些搜索操作，参考附录 F 可以获得更多相关信息。

6.7.6 优化 SQL_CALC_FOUND_ROWS

248

分页的时候，另一个常用的技巧是在 LIMIT 语句中加上 SQL_CALC_FOUND_ROWS 提示（hint），这样就可以获得去掉 LIMIT 以后满足条件的行数，因此可以作为分页的总数。看起来，MySQL 做了一些非常"高深"的优化，像是通过某种方法预测了总行数。但实际上，MySQL 只有在扫描了所有满足条件的行以后，才会知道行数，所以加上这个提示以后，不管是否需要，MySQL 都会扫描所有满足条件的行，然后再抛弃掉不需要的行，而不是在满足 LIMIT 的行数后就终止扫描。所以该提示的代价可能非常高。

一个更好的设计是将具体的页数换成"下一页"按钮，假设每页显示 20 条记录，那么我们每次查询时都是用 LIMIT 返回 21 条记录并只显示 20 条，如果第 21 条存在，那么我们就显示"下一页"按钮，否则就说明没有更多的数据，也就无须显示"下一页"按钮了。

另一种做法是先获取并缓存较多的数据——例如，缓存 1 000 条——然后每次分页都从这个缓存中获取。这样做可以让应用程序根据结果集的大小采取不同的策略，如果结果集少于 1 000，就可以在页面上显示所有的分页链接，因为数据都在缓存中，所以这样做性能不会有问题。如果结果集大于 1 000，则可以在页面上设计一个额外的"找到的结果多于 1 000 条"之类的按钮。这两种策略都比每次生成全部结果集再抛弃掉不需要的数据的效率要高很多。

有时候也可以考虑使用 EXPLAIN 的结果中的 rows 列的值来作为结果集总数的近似值（实际上 Google 的搜索结果总数也是个近似值）。当需要精确结果的时候，再单独使用 COUNT(*) 来满足需求，这时如果能够使用索引覆盖扫描则通常也会比 SQL_CALC_FOUND_ROWS 快得多。

6.7.7 优化 UNION 查询

MySQL 总是通过创建并填充临时表的方式来执行 UNION 查询。因此很多优化策略在 UNION 查询中都没法很好地使用。经常需要手工地将 WHERE、LIMIT、ORDER BY 等子句"下推"到 UNION 的各个子查询中，以便优化器可以充分利用这些条件进行优化（例如，直接将这些子句冗余地写一份到各个子查询）。

除非确实需要服务器消除重复的行，否则就一定要使用 UNION ALL，这一点很重要。如果没有 ALL 关键字，MySQL 会给临时表加上 DISTINCT 选项，这会导致对整个临时表的数据做唯一性检查。这样做的代价非常高。即使有 ALL 关键字，MySQL 仍然会使用临时表存储结果。事实上，MySQL 总是将结果放入临时表，然后再读出，再返回给客户端。虽然很多时候这样做是没有必要的（例如，MySQL 可以直接把这些结果返回给客户端）。

249

6.7.8 静态查询分析

Percona Toolkit 中的 *pt-query-advisor* 能够解析查询日志、分析查询模式，然后给出所有可能存在潜在问题的查询，并给出足够详细的建议。这像是给 MySQL 所有的查询做一次全面的健康检查。它能检测出许多常见的问题，诸如我们前面介绍的内容。

6.7.9 使用用户自定义变量

用户自定义变量是一个容易被遗忘的 MySQL 特性，但是如果能够用好，发挥其潜力，在某些场景可以写出非常高效的查询语句。在查询中混合使用过程化和关系化逻辑的时候，自定义变量可能会非常有用。单纯的关系查询将所有的东西都当成无序的数据集合，并且一次性操作它们。MySQL 则采用了更加程序化的处理方式。MySQL 的这种方式有它的弱点，但如果能熟练地掌握，则会发现其强大之处，而用户自定义变量也可以给这种方式带来很大的帮助。

用户自定义变量是一个用来存储内容的临时容器，在连接 MySQL 的整个过程中都存在。可以使用下面的 SET 和 SELECT 语句来定义它们[注28]：

```
mysql> SET @one       := 1;
mysql> SET @min_actor := (SELECT MIN(actor_id) FROM sakila.actor);
mysql> SET @last_week := CURRENT_DATE-INTERVAL 1 WEEK;
```

然后可以在任何可以使用表达式的地方使用这些自定义变量：

```
mysql> SELECT ... WHERE col <= @last_week;
```

在了解自定义变量的强大之前，我们再看看它自身的一些属性和限制，看看在哪些场景下我们不能使用用户自定义变量：

- 使用自定义变量的查询，无法使用查询缓存。
- 不能在使用常量或者标识符的地方使用自定义变量，例如表名、列名和 LIMIT 子句中。
- 用户自定义变量的生命周期是在一个连接中有效，所以不能用它们来做连接间的通信。
- 如果使用连接池或者持久化连接，自定义变量可能让看起来毫无关系的代码发生交互（如果是这样，通常是代码 bug 或者连接池 bug，这类情况确实可能发生）。
- 在 5.0 之前的版本，是大小写敏感的，所以要注意代码在不同 MySQL 版本间的兼容性问题。
- 不能显式地声明自定义变量的类型。确定未定义变量的具体类型的时机在不同

注 28：在某些场景下，也可以直接使用 = 进行赋值，不过为了避免歧义，建议始终使用 :=。

MySQL 版本中也可能不一样。如果你希望变量是整数类型，那么最好在初始化的时候就赋值为 0，如果希望是浮点型则赋值为 0.0，如果希望是字符串则赋值为 ''，用户自定义变量的类型在赋值的时候会改变。MySQL 的用户自定义变量是一个动态类型。 250

- MySQL 优化器在某些场景下可能会将这些变量优化掉，这可能导致代码不按预想的方式运行。
- 赋值的顺序和赋值的时间点并不总是固定的，这依赖于优化器的决定。实际情况可能很让人困惑，后面我们将看到这一点。
- 赋值符号 := 的优先级非常低，所以需要注意，赋值表达式应该使用明确的括号。
- 使用未定义变量不会产生任何语法错误，如果没有意识到这一点，非常容易犯错。

优化排名语句

使用用户自定义变量[注29]的一个重要特性是你可以在给一个变量赋值的同时使用这个变量。换句话说，用户自定义变量的赋值具有“左值”特性。下面的例子展示了如何使用变量来实现一个类似“行号（row number）”的功能：

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS rownum
    -> FROM sakila.actor LIMIT 3;
+----------+--------+
| actor_id | rownum |
+----------+--------+
|        1 |      1 |
|        2 |      2 |
|        3 |      3 |
+----------+--------+
```

这个例子的实际意义并不大，它只是实现了一个和该表主键一样的列。不过，我们也可以把这当作一个排名。现在我们来看一个更复杂的用法。我们先编写一个查询获取演过最多电影的前 10 位演员，然后根据他们的出演电影次数做一个排名，如果出演的电影数量一样，则排名相同。我们先编写一个查询，返回每个演员参演电影的数量：

```
mysql> SELECT actor_id, COUNT(*) as cnt
    -> FROM sakila.film_actor
    -> GROUP BY actor_id
    -> ORDER BY cnt DESC
    -> LIMIT 10;
+----------+-----+
| actor_id | cnt |
+----------+-----+
|      107 |  42 |
|      102 |  41 |
|      198 |  40 |
```

注 29： 为行文方便，后面在不引起歧义的情况下将简称为“变量”。——译者注

```
|      181 |  39 |
|       23 |  37 |
|       81 |  36 |
|      106 |  35 |
|       60 |  35 |
|       13 |  35 |
|      158 |  35 |
+----------+-----+
```

251

现在我们再把排名加上去，这里看到有四名演员都参演了35部电影，所以他们的排名应该是相同的。我们使用三个变量来实现：一个用来记录当前的排名，一个用来记录前一个演员的排名，还有一个用来记录当前演员参演的电影数量。只有当前演员参演的电影的数量和前一个演员不同时，排名才变化。我们先试试下面的写法：

```
mysql> SET @curr_cnt := 0, @prev_cnt := 0, @rank := 0;
mysql> SELECT actor_id,
    ->    @curr_cnt := COUNT(*) AS cnt,
    ->    @rank     := IF(@prev_cnt <> @curr_cnt, @rank + 1, @rank) AS rank,
    ->    @prev_cnt := @curr_cnt AS dummy
    -> FROM sakila.film_actor
    -> GROUP BY actor_id
    -> ORDER BY cnt DESC
    -> LIMIT 10;
+----------+-----+------+-------+
| actor_id | cnt | rank | dummy |
+----------+-----+------+-------+
|      107 |  42 |    0 |     0 |
|      102 |  41 |    0 |     0 |
...
```

Oops——排名和统计列一直都无法更新，这是什么原因？

对这类问题，是没法给出一个放之四海皆准的答案的，例如，一个变量名的拼写错误就可能导致这样的问题（这个案例中并不是这个原因），具体问题要具体分析。这里，通过EXPLAIN我们看到将会使用临时表和文件排序，所以可能是由于变量赋值的时间和我们预料的不同。

在使用用户自定义变量的时候，经常会遇到一些“诡异”的现象，要揪出这些问题的原因通常都不容易，但是相比其带来的好处，深究这些问题是值得的。使用SQL语句生成排名值通常需要做两次计算，例如，需要额外计算一次出演过相同数量电影的演员有哪些。使用变量则可一次完成——这对性能是一个很大的提升。

针对这个案例，另一个简单的方案是在FROM子句中使用子查询生成一个中间的临时表：

```
mysql> SET @curr_cnt := 0, @prev_cnt := 0, @rank := 0;
    -> SELECT actor_id,
    ->    @curr_cnt := cnt AS cnt,
    ->    @rank     := IF(@prev_cnt <> @curr_cnt, @rank + 1, @rank) AS rank,
    ->    @prev_cnt := @curr_cnt AS dummy
    -> FROM (
    ->    SELECT actor_id, COUNT(*) AS cnt
    ->    FROM sakila.film_actor
    ->    GROUP BY actor_id
    ->    ORDER BY cnt DESC
    ->    LIMIT 10
    -> ) as der;
+----------+-----+------+-------+
| actor_id | cnt | rank | dummy |
+----------+-----+------+-------+
|      107 |  42 |    1 |    42 |
|      102 |  41 |    2 |    41 |
|      198 |  40 |    3 |    40 |
|      181 |  39 |    4 |    39 |
|       23 |  37 |    5 |    37 |
|       81 |  36 |    6 |    36 |
|      106 |  35 |    7 |    35 |
|       60 |  35 |    7 |    35 |
|       13 |  35 |    7 |    35 |
|      158 |  35 |    7 |    35 |
+----------+-----+------+-------+
```

252

避免重复查询刚刚更新的数据

如果在更新行的同时又希望获得该行的信息，要怎么做才能避免重复的查询呢？不幸的是，MySQL 并不支持像 PostgreSQL 那样的 UPDATE RETURNING 语法，这个语法可以帮你在更新行的时候同时返回该行的信息。还好在 MySQL 中你可以使用变量来解决这个问题。例如，我们的一个客户希望能够更高效地更新一条记录的时间戳，同时希望查询当前记录中存放的时间戳是什么。简单地，可以用下面的代码来实现：

```
UPDATE t1 SET lastUpdated = NOW() WHERE id = 1;
SELECT lastUpdated FROM t1 WHERE id = 1;
```

使用变量，我们可以按如下方式重写查询：

```
UPDATE t1 SET lastUpdated = NOW() WHERE id = 1 AND @now := NOW();
SELECT @now;
```

上面看起来仍然需要两个查询，需要两次网络来回，但是这里的第二个查询无须访问任何数据表，所以会快非常多。（如果网络延迟非常大，那么这个优化的意义可能不大，不过对这个客户，这样做的效果很好。）

统计更新和插入的数量

当使用了 INSERT ON DUPLICATE KEY UPDATE 的时候，如果想知道到底插入了多少行数据，到底有多少数据是因为冲突而改写成更新操作的？ Kerstian Köhntopp 在他的博客上给出了一个解决这个问题的办法[注30]。实现办法的本质如下：

```
INSERT INTO t1(c1, c2) VALUES(4, 4), (2, 1), (3, 1)
ON DUPLICATE KEY UPDATE
   c1 = VALUES(c1) + ( 0 * ( @x := @x +1 ) );
```

253 当每次由于冲突导致更新时对变量 @x 自增一次。然后通过对这个表达式乘以 0 来让其不影响要更新的内容。另外，MySQL 的协议会返回被更改的总行数，所以不需要单独统计这个值。

确定取值的顺序

使用用户自定义变量的一个最常见的问题就是没有注意到在赋值和读取变量的时候可能是在查询的不同阶段。例如，在 SELECT 子句中进行赋值然后在 WHERE 子句中读取变量，则可能变量取值并不如你所想。下面的查询看起来只返回一个结果，但事实并非如此：

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS cnt
    -> FROM sakila.actor
    -> WHERE @rownum <= 1;
+----------+------+
| actor_id | cnt  |
+----------+------+
|        1 |    1 |
|        2 |    2 |
+----------+------+
```

因为 WHERE 和 SELECT 是在查询执行的不同阶段被执行的。如果在查询中再加入 ORDER BY 的话，结果可能会更不同：

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS cnt
    -> FROM sakila.actor
    -> WHERE @rownum <= 1
    -> ORDER BY first_name;
```

这是因为 ORDER BY 引入了文件排序，而 WHERE 条件是在文件排序操作之前取值的，所以这条查询会返回表中的全部记录。解决这个问题的办法是让变量的赋值和取值发生在执行查询的同一阶段：

注 30： 参考 *http://mysqldump.azundris.com/archives/86-Down-the-dirty-road.html*。

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum AS rownum
    -> FROM sakila.actor
    -> WHERE (@rownum := @rownum + 1) <= 1;
+----------+--------+
| actor_id | rownum |
+----------+--------+
|        1 | 1      |
+----------+--------+
```

小测试：如果在上面的查询中再加上 ORDER BY，那会返回什么结果？试试看吧。如果得出的结果出乎你的意料，想想为什么？再看下面这个查询会返回什么，下面的查询中 ORDER BY 子句会改变变量值，那 WHERE 语句执行时变量值是多少。

254

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, first_name, @rownum AS rownum
    -> FROM sakila.actor
    -> WHERE @rownum <= 1
    -> ORDER BY first_name, LEAST(0, @rownum := @rownum + 1);
```

这个最出人意料的变量行为的答案可以在 EXPLAIN 语句中找到，注意看在 Extra 列中的“Using where”、“Using temporary”或者“Using filesort”。

在上面的最后一个例子中，我们引入了一个新的技巧：我们将赋值语句放到 LEAST() 函数中，这样就可以在完全不改变排序顺序的时候完成赋值操作（在上面例子中，LEAST() 函数总是返回 0）。这个技巧在不希望对子句的执行结果有影响却又要完成变量赋值的时候很有用。这个例子中，无须在返回值中新增额外列。这样的函数还有 GREATEST()、LENGHT()、ISNULL()、NULLIFL()、IF() 和 COALESCE()，可以单独使用也可以组合使用。例如，COALESCE() 可以在一组参数中取第一个已经被定义的变量。

编写偷懒的 UNION

假设需要编写一个 UNION 查询，其第一个子查询作为分支条件先执行，如果找到了匹配的行，则跳过第二个分支。在某些业务场景中确实会有这样的需求，比如先在一个频繁访问的表中查找“热”数据，找不到再去另外一个较少访问的表中查找“冷”数据。（区分热数据和冷数据是一个很好的提高缓存命中率的办法）。

下面的查询会在两个地方查找一个用户——一个主用户表、一个长时间不活跃的用户表，不活跃用户表的目的是为了实现更高效的归档[注 31]：

注 31：Baron 认为在一些社交网站上归档一些常见不活跃用户后，用户重新回到网站时有这样的需求，当用户再次登录时，一方面我们需要将其从归档中重新拿出来，另外，还可以给他发送一份欢迎邮件。这对一些不活跃的用户是非常好的一个优化。在第 11 章我们还会再次讨论这个问题。

```
SELECT id FROM users WHERE id = 123
UNION ALL
SELECT id FROM users_archived WHERE id = 123;
```

上面这个查询是可以正常工作的，但是即使在 users 表中已经找到了记录，上面的查询还是会去归档表 users_archived 中再查找一次。我们可以用一个偷懒的 UNION 查询来抑制这样的数据返回，而且只有当第一个表中没有数据时，我们才在第二个表中查询。一旦在第一个表中找到记录，我们就定义一个变量 @found。我们通过在结果列中做一次赋值来实现，然后将赋值放在函数 GREATEST 中来避免返回额外的数据。为了明确我们的结果到底来自哪个表，我们新增了一个包含表名的列。最后我们需要在查询的末尾将变量重置为 NULL，这样保证遍历时不干扰后面的结果。完成的查询如下：

255

```
SELECT GREATEST(@found := -1, id) AS id, 'users' AS which_tbl
FROM users WHERE id = 1
UNION ALL
  SELECT id, 'users_archived'
  FROM users_archived WHERE id = 1 AND @found IS NULL
UNION ALL
  SELECT 1, 'reset' FROM DUAL WHERE ( @found := NULL ) IS NOT NULL;
```

用户自定义变量的其他用处

不仅是在 SELECT 语句中，在其他任何类型的 SQL 语句中都可以对变量进行赋值。事实上，这也是用户自定义变量最大的用途。例如，可以像前面使用子查询的方式改进排名语句一样来改进 UPDATE 语句。

不过，我们需要使用一些技巧来获得我们希望的结果。有时，优化器会把变量当作一个编译时常量来对待，而不是对其进行赋值。将函数放在类似于 LEAST() 这样的函数中通常可以避免这样的问题。另一个办法是在查询被执行前检查变量是否被赋值。不同的场景下使用不同的办法。

通过一些实践，可以了解所有用户自定义变量能够做的有趣的事情，例如下面这些用法：

- 查询运行时计算总数和平均值。
- 模拟 GROUP 语句中的函数 FIRST() 和 LAST()。
- 对大量数据做一些数据计算。
- 计算一个大表的 MD5 散列值。
- 编写一个样本处理函数，当样本中的数值超过某个边界值的时候将其变成 0。
- 模拟读 / 写游标。
- 在 SHOW 语句的 WHERE 子句中加入变量值。

C.J. DATE 的难题

C.J.DATE 建议在使用数据库设计方法时尽量让 SQL 数据库符合传统关系数据库的要求。这也是根据关系模型设计 SQL 时的初衷，但坦白地说，在这一点上，MySQL 远不如其他数据库管理系统做得好。所以如果按照 C.J. DATE 书中的建议编写的适合关系模型的 SQL 语句在 MySQL 中运行的效率并不高，例如编写一个多层的子查询。很不幸，这是因为 MySQL 本身的限制导致无法按照标准的模式运行。我们强烈建议你阅读这本书 *SQL and Relational Theory: How to Write Accurate SQL Code*（*http://shop.xreilly.com/product/0636920022879.do*）（O'Reilly 出版），它将改变你对 SQL 语句的认识。

6.8 案例学习 256

通常，我们要做的不是查询优化，不是库表结构优化，不是索引优化也不是应用设计优化——在实践中可能要面对所有这些搅和在一起的情况。本节的案例将为大家介绍一些经常困扰用户的问题和解决方法。另外我们还要推荐 Bill Karwin 的书 *SQL Antipatterns*（一本实践型的书籍）。它将介绍如何使用 SQL 解决各种程序员疑难杂症。

6.8.1 使用 MySQL 构建一个队列表

使用 MySQL 来实现队列表是一个取巧的做法，我们看到很多系统在高流量、高并发的情况下表现并不好。典型的模式是一个表包含多种类型的记录：未处理记录、已处理记录、正在处理记录等。一个或者多个消费者线程在表中查找未处理的记录，然后声称正在处理，当处理完成后，再将记录更新成已处理状态。一般的，例如邮件发送、多命令处理、评论修改等会使用类似模式。

通常有两个原因使得大家认为这样的处理方式并不合适。第一，随着队列表越来越大和索引深度的增加，找到未处理记录的速度会随之变慢。你可以通过将队列表分成两部分来解决这个问题，就是将已处理记录归档或者存放到历史表，这可以始终保证队列表很小。

第二，一般的处理过程分两步，先找到未处理记录然后加锁。找到记录会增加服务器的压力，而加锁操作则会让各个消费者进程增加竞争，因为这是一个串行化的操作。在第 11 章，我们会看到这为什么会限制可扩展性。

找到未处理记录一般来说都没问题，如果有问题则可以通过使用消息的方式来通知各个消费者。具体的，可以使用一个带有注释的 SLEEP() 函数做超时处理，如下：

```
SELECT /* waiting on unsent_emails */ SLEEP(10000);
```

这让线程一直阻塞，直到两个条件之一满足：10 000 秒后超时，或者另一个线程使用 KILL QUERY 结束当前的 SLEEP。因此，当再向队列表中新增一批数据后，可以通过 SHOW PROCESSLIST，根据注释找到当前正在休眠的线程，并将其 KILL。你可以使用函数 GET_LOCK() 和 RELEASE_LOCK() 来实现通知，或者可以在数据库之外实现，例如使用一个消息服务。

最后需要解决的问题是如何让消费者标记正在处理的记录，而不至于让多个消费者重复处理一个记录。我们看到大家一般使用 SELECT FOR UPDATE 来实现。这通常是扩展性问题的根源，这会导致大量的事务阻塞并等待。

257 一般，我们要尽量避免使用 SELECT FOR UPDATE。不光是队列表，任何情况下都要尽量避免。总是有别的更好的办法实现你的目的。在队列表的案例中，可以直接使用 UPDATE 来更新记录，然后检查是否还有其他的记录需要处理。我们看看具体实现，我们先建立如下的表：

```
CREATE TABLE unsent_emails (
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  -- columns for the message, from, to, subject, etc.
  status ENUM('unsent', 'claimed', 'sent'),
  owner  INT UNSIGNED NOT NULL DEFAULT 0,
  ts     TIMESTAMP,
  KEY    (owner, status, ts)
);
```

该表的列 owner 用来存储当前正在处理这个记录的连接 ID，即由函数 CONNECTION_ID() 返回的 ID。如果当前记录没有被任何消费者处理，则该值为 0。

我们还经常看到的一个办法是，如下面所示的一次处理 10 条记录：

```
BEGIN;
SELECT id FROM unsent_emails
   WHERE owner = 0 AND status = 'unsent'
   LIMIT 10 FOR UPDATE;
-- result: 123, 456, 789
UPDATE unsent_emails
   SET status = 'claimed', owner = CONNECTION_ID()
   WHERE id IN(123, 456, 789);
COMMIT;
```

看到这里的 SELECT 查询可以使用到索引的两个列，因此理论上查找的效率应该更快。问题是，在上面两个查询之间的“间隙时间”，这里的锁会让所有其他同样的查询全部都

被阻塞。所有的这样的查询将使用相同的索引，扫描索引相同的部分，所以很可能会被阻塞。

如果改进成下面的写法，则会更加高效：

```
SET AUTOCOMMIT = 1;
COMMIT;
UPDATE unsent_emails
   SET status = 'claimed', owner = CONNECTION_ID()
   WHERE owner = 0 AND status = 'unsent'
   LIMIT 10;
SET AUTOCOMMIT = 0;
SELECT id FROM unsent_emails
   WHERE owner = CONNECTION_ID() AND status = 'claimed';
-- result: 123, 456, 789
```

根本就无须使用 SELECT 查询去找到哪些记录还没有被处理。客户端的协议会告诉你更新了几条记录，所以可以知道这次需要处理多少条记录。

所有的 SELECT FOR UPDATE 都可以使用类似的方法改写。 258

最后还需要处理一种特殊情况：那些正在被进程处理，而进程本身却由于某种原因退出的情况。这种情况处理起来很简单。你只需要定期运行 UPDATE 语句将它都更新成原始状态就可以了，然后执行 SHOW PROCESSLIST，获取当前正在工作的线程 ID，并使用一些 WHERE 条件避免取到那些刚开始处理的进程。假设我们获取的线程 ID 有（10、20、30），下面的更新语句会将处理时间超过 10 分钟的记录状态都更新成初始状态：

```
UPDATE unsent_emails
   SET owner = 0, status = 'unsent'
   WHERE owner NOT IN(0, 10, 20, 30) AND status = 'claimed'
   AND ts < CURRENT_TIMESTAMP - INTERVAL 10 MINUTE;
```

另外，注意看看是如何巧妙地设计索引让这个查询更加高效的。这也是上一章和本章知识的结合。因为我们将范围条件放在 WHERE 条件的末尾，这个查询恰好能够使用索引的全部列。其他的查询也都能用上这个索引，这就避免了再新增一个额外的索引来满足其他的查询。

这里我们将总结一下这个案例中的一些基础原则：

- 尽量少做事，可以的话就不要做任何事情。除非不得已，否则不要使用轮询，因为这会增加负载，而且还会带来很多低产出的工作。
- 尽可能快地完成需要做的事情。尽量使用 UPDATE 代替先 SELECT FOR UPDATE 再 UPDATE 的写法，因为事务提交的速度越快，持有的锁时间就越短，可以大大减少竞争和加速串行执行效率。将已经处理完成和未处理的数据分开，保证数据集足够小。

- 这个案例的另一个启发是，某些查询是无法优化的；考虑使用不同的查询或者不同的策略去实现相同的目的。通常对于 SELECT FOR UPDATE 就需要这样处理。

有时，最好的办法就是将任务队列从数据库中迁移出来。Redis 就是一个很好的队列容器，也可以使用 *memcached* 来实现。另一个选择是使用 Q4M 存储引擎，但我们没有在生产环境使用过这个存储引擎，所以这里也没办法提供更多的参考。RabbitMQ 和 Gearman[注 32] 也可以实现类似的功能。

6.8.2 计算两点之间的距离

地理信息计算再次出现在我们的书中了。不建议用户使用 MySQL 做太复杂的空间信息
259 存储——PostgreSQL 在这方面是不错的选择——我们这里将介绍一些常用的计算模式。一个典型的例子是计算以某个点为中心，一定半径内的所有点。

典型的实际案例可能是查找某个点附近所有可以出租的房子，或者社交网站中“匹配”附近的用户，等等。假设我们有如下表：

```
CREATE TABLE locations (
  id   INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(30),
  lat  FLOAT NOT NULL,
  lon  FLOAT NOT NULL
);
INSERT INTO locations(name, lat, lon)
  VALUES('Charlottesville, Virginia', 38.03, -78.48),
        ('Chicago, Illinois',         41.85, -87.65),
        ('Washington, DC',            38.89, -77.04);
```

这里经度和纬度的单位是“度”，通常我们假设地球是圆的，然后使用两点所在最大圆（半正矢）公式来计算两点之间的距离。现在有坐标 *latA* 和 *lonA*、*latB* 和 *lonB*，那么点 A 和点 B 的距离计算公式如下：

```
ACOS(
   COS(latA) * COS(latB) * COS(lonA - lonB)
     + SIN(latA) * SIN(latB)
)
```

计算出的结果是一个弧度，如果要将结果的单位转换成英里或者千米，则需要乘以地球的半径，也就是 3 959 英里或者 6 371 千米。假设我们需要找出所有距离 Baron 所居住的地方 Charlottesville 100 英里以内的点，那么我们需要将经纬度带入上面的计算公式：

注 32： 参考 *http://www.rabbitmq.com* 和 *http://gearman.org*。

```
SELECT * FROM locations WHERE 3979 * ACOS(
   COS(RADIANS(lat)) * COS(RADIANS(38.03)) * COS(RADIANS(lon) - RADIANS(-78.48))
     + SIN(RADIANS(lat)) * SIN(RADIANS(38.03))
) <= 100;
+----+---------------------------+-------+--------+
| id | name                      | lat   | lon    |
+----+---------------------------+-------+--------+
|  1 | Charlottesville, Virginia | 38.03 | -78.48 |
|  3 | Washington, DC            | 38.89 | -77.04 |
+----+---------------------------+-------+--------+
```

这类查询不仅无法使用索引，而且还会非常消耗 CPU 时间，给服务器带来很大的压力，而且我们还得反复计算这个。那要怎样优化呢？

这个设计中有几个地方可以做优化。第一，看看是否真的需要这么精确的计算。其实这种算法已经有很多不精确的地方了，如下所示：

- 两个地方之间的直线距离可能是 100 英里，但实际上它们之间的行走距离很可能不是这个值。无论你们在哪两个地方，要到达彼此位置的行走距离多半都不是直线距离，路上可能需要绕很多的弯，比如说如果有一条河，需要绕远走到一个有桥的地方。所以，这里计算的绝对距离只是一个参考值。 260
- 如果我们根据邮政编码来确定某个人所在的地区，再根据这个地区的中心位置计算他和别人的距离，那么这本身就是一个估算。Baron 住在 Charlottesville，不过不是在中心地区，他对华盛顿物理位置的中心也不感兴趣。

所以，通常并不需要精确计算，很多应用如果这样计算，多半是认真过头了。这类似于有效数字的估算：计算结果的精度永远都不会比测量的值更高。（换句话说，“错进，错出”。）

如果不需要太高的精度，那么我们认为地球是圆的应该也没什么问题，其实准确的说应该是椭圆。根据毕达哥拉斯定理，做些三角函数变换，我们可以把上面的公式转换得更简单，只需要做些求和、乘积以及平方根运算，就可以得出一个点是否在另一个点多少英里之内。[注 33]

等等，为什么就到这为止？我们是否真需要计算一个圆周呢？为什么不直接使用一个正方形代替？边长为 200 英里的正方形，一个顶点到中心的距离大概是 141 英里，这和实际计算的 100 英里相差得并不是那么远。那我们根据正方形公式来计算弧度为 0.0253（100 英里）的中心到边长的距离：

注 33：要想有更多的优化，你可以将三角函数的计算放到应用中，而不要在数据库中计算。三角函数是非常消耗 CPU 的操作。如果将坐标都转换成弧度存放，则对数据库来说就简化了很多。为了保证我们的案例简单，不要引入太多别的因子，所以这里我们将不再做更多的优化了。

```
SELECT * FROM locations
WHERE lat BETWEEN  38.03 - DEGREES(0.0253) AND  38.03 + DEGREES(0.0253)
  AND lon BETWEEN -78.48 - DEGREES(0.0253) AND -78.48 + DEGREES(0.0253);
```

现在我们看看如何使用索引来优化这个查询。简单地，我们可以增加索引 (lat,lon) 或者 (lon,lat)。不过这样做效果并不会很好。正如我们所知，MySQL 5.5 和之前的版本，如果第一列是范围查询的话，就无法使用索引后面的列了。因为两个列都是范围的，所以这里只能使用索引的一个列（BETWEEN 等效于一个大于和一个小于）。

我们再次想起了通常使用的 IN() 优化。我们先新增两个列，用来存储坐标的近似值 FLOOR()，然后在查询中使用 IN() 将所有点的整数值都放到列表中。下面是我们需要新增的列和索引：

261

```
mysql> ALTER TABLE locations
    ->    ADD lat_floor INT NOT NULL DEFAULT 0,
    ->    ADD lon_floor INT NOT NULL DEFAULT 0,
    ->    ADD KEY(lat_floor, lon_floor);
mysql> UPDATE locations
    -> SET lat_floor = FLOOR(lat), lon_floor = FLOOR(lon);
```

现在我们可以根据坐标的一定范围的近似值来搜索了，这个近似值包括地板值和天花板值，地理上分别对应的是南北。下面的查询为我们只展示了如何查某个范围的所有点；数值需要在应用程序中计算而不是 MySQL 中：

```
mysql> SELECT FLOOR( 38.03 - DEGREES(0.0253)) AS lat_lb,
    ->      CEILING( 38.03 + DEGREES(0.0253)) AS lat_ub,
    ->        FLOOR(-78.48 - DEGREES(0.0253)) AS lon_lb,
    ->      CEILING(-78.48 + DEGREES(0.0253)) AS lon_ub;
+--------+--------+--------+--------+
| lat_lb | lat_ub | lon_lb | lon_ub |
+--------+--------+--------+--------+
|     36 |     40 |    -80 |    -77 |
+--------+--------+--------+--------+
```

现在我们就可以生成 IN() 列表中的整数了，也就是前面计算的地板和天花板数值之间的数字。下面是加上 WHERE 条件的完整查询：

```
SELECT * FROM locations
WHERE lat BETWEEN  38.03 - DEGREES(0.0253) AND  38.03 + DEGREES(0.0253)
  AND lon BETWEEN -78.48 - DEGREES(0.0253) AND -78.48 + DEGREES(0.0253)
  AND lat_floor IN(36,37,38,39,40) AND lon_floor IN(-80,-79,-78,-77);
```

使用近似值会让我们的计算结果有些偏差，所以我们还需要一些额外的条件剔除在正方形之外的点。这和前面使用 CRC32 做哈希索引类似：先建一个索引帮我们过滤出近似值，再使用精确条件匹配所有的记录并移除不满足条件的记录。

事实上，到这时我们就无须根据正方形的近似来过滤数据了，我们可以使用最大圆公式

或者毕达哥拉斯定理来计算：

```
SELECT * FROM locations
WHERE lat_floor IN(36,37,38,39,40) AND lon_floor IN(-80,-79,-78,-77)
 AND 3979 * ACOS(
   COS(RADIANS(lat)) * COS(RADIANS(38.03)) * COS(RADIANS(lon) - RADIANS(-78.48))
     + SIN(RADIANS(lat)) * SIN(RADIANS(38.03))
  ) <= 100;
```

这时计算精度再次回到前面——使用一个精确的圆周——不过，现在的做法更快[注34]。只要能够高效地过滤掉大部分的点，例如使用近似整数和索引，之后再做精确数学计算的代价并不大。只是不要直接使用大圆周的算法，否则速度会很慢。

> Sphinx 有很多内置的地理信息搜索功能，比 MySQL 实现要好很多。如果正在考虑使用 MyISAM 的 GIS 函数，并使用上面的技巧来计算，那么你需要记住：这样做效果并不会很好，MyISAM 本身也并不适合大数据量、高并发的应用，另外 MyISAM 本身还有一些弱点，如数据文件崩溃、表级锁等。 262

回顾一下上面的案例，我们采用了下面这些常用的优化策略：

- 尽量少做事，可能的话尽量不做事。这个案例中就不要对所有的点计算大圆周公式；先使用简单的方案过滤大多数数据，然后再到过滤出来的更小的集合上使用复杂的公式运算。
- 快速地完成事情。确保在你的设计中尽可能地让查询都用上合适的索引，使用近似计算（例如本案例中，认为地球是平的，使用一个正方形来近似圆周）来避免复杂的计算。
- 需要的时候，尽可能让应用程序完成一些计算。例如本案例中，在应用程序中计算所有的三角函数。

6.8.3 使用用户自定义函数

当 SQL 语句已经无法高效地完成某些任务的时候，这里我们将介绍最后一个高级的优化技巧。当你需要更快的速度，那么 C 和 C++ 是很好的选择。当然，你需要一定的 C 或 C++ 编程技巧，否则你写的程序很可能会让服务器崩溃。这和“能力越强，责任越大”类似。

我们将在下一章为你展示如何编写一个用户自定义函数（UDFs），不过这一章就将通过一个案例看看如何用好一个用户自定义函数。有一个客户，在项目中需要如下的功能：“我们需要根据两个随机的 64 位数字计算它们的 XOR 值，来看两个数值是否匹配。大约有 3 500 万条的记录需要在秒级别完成。”经过简单的计算就知道，当前的硬件条件下，不

注 34： 再一次，需要使用应用程序中的代码来计算这样的表达式 COS(RADIANS(38.03))。

可能在 MySQL 中完成。那如何解决这个问题呢？

问题的答案是使用 Yves Trudeau 编写的一个计算程序，这个程序使用 SSE4.2 指令集，以一个后台程序的方式运行在通用服务器上，然后我们编写一个用户自定义函数，通过简单的网络通信协议和前面的程序进行交互。

Yves 的测试表明，分布式运行上面的程序，可以达到在 130 毫秒内完成 4 百万次匹配计算。通过这样的方式，可以将密集型的计算放到一些通用的服务器上，同时可以对外界完全透明，看起来是 MySQL 完成了全部的工作。正如他们在 Twitter 上说的：# 太好了！这是一个典型的业务优化案例，而不仅仅是优化了一个简单的技术问题。

263

6.9 总结

如果把创建高性能应用程序比作是一个环环相扣的“难题”，除了前面介绍的 schema、索引和查询语句设计之外，查询优化应该是解开“难题”的最后一步了。要想写一个好的查询，你必须要理解 schema 设计、索引设计等，反之亦然。

理解查询是如何被执行的以及时间都消耗在哪些地方，这依然是前面我们介绍的响应时间的一部分。再加上一些诸如解析和优化过程的知识，就可以更进一步地理解上一章讨论的 MySQL 如何访问表和索引的内容了。这也从另一个维度帮助读者理解 MySQL 在访问表和索引时查询和索引的关系。

优化通常都需要三管齐下：不做、少做、快速地做。我们希望这里的案例能够帮助你将理论和实践联系起来。

除了这些基础的手段，包括查询、表结构、索引等，MySQL 还有一些高级的特性可以帮助你优化应用，例如分区，分区和索引有些类似但是原理不同。MySQL 还支持查询缓存，它可以帮你缓存查询结果，当完全相同的查询再次执行时，直接使用缓存结果（回想一下，“不做”）。我们将在下一章中介绍这些特性。

第 7 章 <265>

MySQL高级特性

MySQL 从 5.0 和 5.1 版本开始引入了很多高级特性，例如分区、触发器等，这对有其他关系型数据库使用背景的用户来说可能并不陌生。这些新特性吸引了很多用户开始使用 MySQL。不过，这些特性的性能到底如何，还需要用户真正使用过才能知道。本章我们将为大家介绍，在真实的世界中，这些特性表现如何，而不是只简单地介绍参考手册或者宣传材料上的数据。

7.1 分区表

对用户来说，分区表是一个独立的逻辑表，但是底层由多个物理子表组成。实现分区的代码实际上是对一组底层表的句柄对象（Handler Object）的封装。对分区表的请求，都会通过句柄对象转化成对存储引擎的接口调用。所以分区对于 SQL 层来说是一个完全封装底层实现的黑盒子，对应用是透明的，但是从底层的文件系统来看就很容易发现，每一个分区表都有一个使用 # 分隔命名的表文件。

MySQL 实现分区表的方式——对底层表的封装——意味着索引也是按照分区的子表定义的，而没有全局索引。这和 Oracle 不同，在 Oracle 中可以更加灵活地定义索引和表是否进行分区。

MySQL 在创建表时使用 `PARTITION BY` 子句定义每个分区存放的数据。在执行查询的时候，优化器会根据分区定义过滤那些没有我们需要数据的分区，这样查询就无须扫描所有分区——只需要查找包含需要数据的分区就可以了。

分区的一个主要目的是将数据按照一个较粗的粒度分在不同的表中。这样做可以将相关的数据存放在一起，另外，如果想一次批量删除整个分区的数据也会变得很方便。

在下面的场景中，分区可以起到非常大的作用： <266>

- 表非常大以至于无法全部都放在内存中，或者只在表的最后部分有热点数据，其他均是历史数据。
- 分区表的数据更容易维护。例如，想批量删除大量数据可以使用清除整个分区的方式。另外，还可以对一个独立分区进行优化、检查、修复等操作。
- 分区表的数据可以分布在不同的物理设备上，从而高效地利用多个硬件设备。
- 可以使用分区表来避免某些特殊的瓶颈，例如 InnoDB 的单个索引的互斥访问、ext3 文件系统的 inode 锁竞争等。
- 如果需要，还可以备份和恢复独立的分区，这在非常大的数据集的场景下效果非常好。

MySQL 的分区实现非常复杂，我们不打算介绍实现的全部细节。这里我们将专注在分区性能方面，所以如果想了解更多的关于分区的基础知识，我们建议阅读 MySQL 官方手册中的“分区”一节，其中介绍了很多分区相关的基础知识。另外，还可以阅读 CREATE TABLE、SHOW CREATE TABLE、ALTER TABLE 和 INFORMATION_SCHEMA.PARTITIONS、EXPLAIN 关于分区部分的介绍。分区特性使得 CREATE TABLE 和 ALTER TABLE 命令变得更加复杂了。

分区表本身也有一些限制，下面是其中比较重要的几点：

- 一个表最多只能有 1024 个分区。
- 在 MySQL 5.1 中，分区表达式必须是整数，或者是返回整数的表达式。在 MySQL 5.5 中，某些场景中可以直接使用列来进行分区。
- 如果分区字段中有主键或者唯一索引的列，那么所有主键列和唯一索引列都必须包含进来。
- 分区表中无法使用外键约束。

7.1.1 分区表的原理

如前所述，分区表由多个相关的底层表实现，这些底层表也是由句柄对象（Handler object）表示，所以我们也可以直接访问各个分区。存储引擎管理分区的各个底层表和管理普通表一样（所有的底层表都必须使用相同的存储引擎），分区表的索引只是在各个底层表上各自加上一个完全相同的索引。从存储引擎的角度来看，底层表和一个普通表没有任何不同，存储引擎也无须知道这是一个普通表还是一个分区表的一部分。

分区表上的操作按照下面的操作逻辑进行：

267 SELECT 查询

当查询一个分区表的时候，分区层先打开并锁住所有的底层表，优化器先判断是否可以过滤部分分区，然后再调用对应的存储引擎接口访问各个分区的数据。

INSERT 操作

当写入一条记录时，分区层先打开并锁住所有的底层表，然后确定哪个分区接收这条记录，再将记录写入对应底层表。

DELETE 操作

当删除一条记录时，分区层先打开并锁住所有的底层表，然后确定数据对应的分区，最后对相应底层表进行删除操作。

UPDATE 操作

当更新一条记录时，分区层先打开并锁住所有的底层表，MySQL 先确定需要更新的记录在哪个分区，然后取出数据并更新，再判断更新后的数据应该放在哪个分区，最后对底层表进行写入操作，并对原数据所在的底层表进行删除操作。

有些操作是支持过滤的。例如，当删除一条记录时，MySQL 需要先找到这条记录，如果 WHERE 条件恰好和分区表达式匹配，就可以将所有不包含这条记录的分区都过滤掉。这对 UPDATE 语句同样有效。如果是 INSERT 操作，则本身就是只命中一个分区，其他分区都会被过滤掉。MySQL 先确定这条记录属于哪个分区，再将记录写入对应的底层分区表，无须对任何其他分区进行操作。

虽然每个操作都会“先打开并锁住所有的底层表”，但这并不是说分区表在处理过程中是锁住全表的。如果存储引擎能够自己实现行级锁，例如 InnoDB，则会在分区层释放对应表锁。这个加锁和解锁过程与普通 InnoDB 上的查询类似。

后面我们会通过一些例子来看看，当访问一个分区表的时候，打开和锁住所有底层表的代价及其带来的后果。

7.1.2 分区表的类型

MySQL 支持多种分区表。我们看到最多的是根据范围进行分区，每个分区存储落在某个范围的记录，分区表达式可以是列，也可以是包含列的表达式。例如，下表就可以将每一年的销售额存放在不同的分区里：

```
CREATE TABLE sales (
   order_date DATETIME NOT NULL,
   -- Other columns omitted
) ENGINE=InnoDB PARTITION BY RANGE(YEAR(order_date)) (
    PARTITION p_2010 VALUES LESS THAN (2010),
    PARTITION p_2011 VALUES LESS THAN (2011),
    PARTITION p_2012 VALUES LESS THAN (2012),
    PARTITION p_catchall VALUES LESS THAN MAXVALUE );
```

268

PARTITION 分区子句中可以使用各种函数。但有一个要求，表达式返回的值要是一个确定的整数，且不能是一个常数。这里我们使用函数 YEAR()，也可以使用任何其他的函数，

如 TO_DAYS()。根据时间间隔进行分区，是一种很常见的分区方式，后面我们还会再回过头来看这个例子，看看如何优化这个例子来避免一些问题。

MySQL 还支持键值、哈希和列表分区，这其中有些还支持子分区，不过我们在生产环境中很少见到。在 MySQL 5.5 中，还可以使用 RANGE COLUMNS 类型的分区，这样即使是基于时间的分区也无须再将其转化成一个整数，后面将详细介绍。

在我们看过的一个子分区的案例中，对一个类似于前面我们设计的按时间分区的 InnoDB 表，系统通过子分区可降低索引的互斥访问的竞争。最近一年的分区的数据会被非常频繁地访问，这会导致大量的互斥量的竞争。使用哈希子分区可以将数据切成多个小片，大大降低互斥量的竞争问题。

我们还看到的一些其他的分区技术包括：

- 根据键值进行分区，来减少 InnoDB 的互斥量竞争。
- 使用数学模函数来进行分区，然后将数据轮询放入不同的分区。例如，可以对日期做模 7 的运算，或者更简单地使用返回周几的函数，如果只想保留最近几天的数据，这样分区很方便。
- 假设表有一个自增的主键列 id，希望根据时间将最近的热点数据集中存放。那么必须将时间戳包含在主键当中才行，而这和主键本身的意义相矛盾。这种情况下也可以使用这样的分区表达式来实现相同的目的：HASH(id DIV 1000000)，这将为 100 万数据建立一个分区。这样一方面实现了当初的分区目的，另一方面比起使用时间范围分区还避免了一个问题，就是当超过一定阈值时，如果使用时间范围分区就必须新增分区。

7.1.3 如何使用分区表

假设我们希望从一个非常大的表中查询出一段时间的记录，而这个表中包含了很多年的历史数据，数据是按照时间排序的，例如，希望查询最近几个月的数据，这大约有 10 亿条记录。可能过些年本书会过时，不过我们还是假设使用的是 2012 年的硬件设备，而原表中有 10TB 的数据，这个数据量远大于内存，并且使用的是传统硬盘，不是闪存（多数 SSD 也没有这么大的空间）。你打算如何查询这个表？如何才能更高效？

269 首先很肯定：因为数据量巨大，肯定不能在每次查询的时候都扫描全表。考虑到索引在空间和维护上的消耗，也不希望使用索引。即使真的使用索引，你会发现数据并不是按照想要的方式聚集的，而且会有大量的碎片产生，最终会导致一个查询产生成千上万的随机 I/O，应用程序也随之僵死。情况好一点的时候，也许可以通过一两个索引解决一些问题。不过多数情况下，索引不会有任何作用。这时候只有两条路可选：让所有的查

询都只在数据表上做顺序扫描，或者将数据表和索引全部都缓存在内存里。

这里需要再陈述一遍：在数据量超大的时候，B-Tree 索引就无法起作用了。除非是索引覆盖查询，否则数据库服务器需要根据索引扫描的结果回表，查询所有符合条件的记录，如果数据量巨大，这将产生大量随机 I/O，随之，数据库的响应时间将大到不可接受的程度。另外，索引维护（磁盘空间、I/O 操作）的代价也非常高。有些系统，如 Infobright，意识到这一点，于是就完全放弃使用 B-Tree 索引，而选择了一些更粗粒度的但消耗更少的方式检索数据，例如在大量数据上只索引对应的一小块元数据。

这正是分区要做的事情。理解分区时还可以将其当作索引的最初形态，以代价非常小的方式定位到需要的数据在哪一片"区域"。在这片"区域"中，你可以做顺序扫描，可以建索引，还可以将数据都缓存到内存，等等。因为分区无须额外的数据结构记录每个分区有哪些数据——分区不需要精确定位每条数据的位置，也就无须额外的数据结构——所以其代价非常低。只需要一个简单的表达式就可以表达每个分区存放的是什么数据。

为了保证大数据量的可扩展性，一般有下面两个策略：

全量扫描数据，不要任何索引。

可以使用简单的分区方式存放表，不要任何索引，根据分区的规则大致定位需要的数据位置。只要能够使用 WHERE 条件，将需要的数据限制在少数分区中，则效率是很高的。当然，也需要做一些简单的运算保证查询的响应时间能够满足需求。使用该策略假设不用将数据完全放入到内存中，同时还假设需要的数据全都在磁盘上，因为内存相对很小，数据很快会被挤出内存，所以缓存起不了任何作用。这个策略适用于以正常的方式访问大量数据的时候。警告：后面我们会详细解释，必须将查询需要扫描的分区个数限制在一个很小的数量。

索引数据，并分离热点。

如果数据有明显的"热点"，而且除了这部分数据，其他数据很少被访问到，那么可以将这部分热点数据单独放在一个分区中，让这个分区的数据能够有机会都缓存在内存中。这样查询就可以只访问一个很小的分区表，能够使用索引，也能够有效地使用缓存。

仅仅知道这些还不够，MySQL 的分区表实现还有很多陷阱。下面我们看看都有哪些，以及如何避免。 270

7.1.4 什么情况下会出问题

上面我们介绍的两个分区策略都基于两个非常重要的假设：查询都能够过滤（prunning）掉很多额外的分区、分区本身并不会带来很多额外的代价。而事实证明，这两个假设在

某些场景下会有问题。下面介绍一些可能会遇到的问题。

NULL 值会使分区过滤无效

关于分区表一个容易让人误解的地方就是分区的表达式的值可以是 NULL：第一个分区是一个特殊分区。假设按照 PARTITION BY RANGE YEAR(order_date) 分区，那么所有 order_date 为 NULL 或者是一个非法值的时候，记录都会被存放到第一个分区[注1]。现在假设有下面的查询：WHERE order_date BETWEEN '2012-01-01' AND '2012-01-31'。实际上，MySQL 会检查两个分区，而不是之前猜想的一个：它会检查 2012 年这个分区，同时它还会检查这个表的第一个分区。检查第一个分区是因为 YEAR() 函数在接收非法值的时候可能会返回 NULL 值，那么这个范围的值可能会返回 NULL 而被存放到第一个分区了。这一点对于其他很多函数，例如 TO_DAYS() 也一样。[注2]

如果第一个分区非常大，特别是当使用“全量扫描数据，不要任何索引”的策略时，代价会非常大。而且扫描两个分区来查找列也不是我们使用分区表的初衷。为了避免这种情况，可以创建一个“无用”的第一个分区，例如，上面的例子中可以使用 PARTITION p_nulls VALUES LESS THAN (0) 来创建第一个分区。如果插入表中的数据都是有效的，那么第一个分区就是空的，这样即使需要检测第一个分区，代价也会非常小。

在 MySQL 5.5 中就不需要这个优化技巧了，因为可以直接使用列本身而不是基于列的函数进行分区：PARTITION BY RANGE COLUMNS(order_date)。所以这个案例最好的解决方法是能够直接使用 MySQL 5.5 的这个语法。

分区列和索引列不匹配

如果定义的索引列和分区列不匹配，会导致查询无法进行分区过滤。假设在列 a 上定义了索引，而在列 b 上进行分区。因为每个分区都有其独立的索引，所以扫描列 b 上的索引就需要扫描每一个分区内对应的索引。如果每个分区内对应索引的非叶子节点都在内存中，那么扫描的速度还可以接受，但如果能跳过某些分区索引当然
271 会更好。要避免这个问题，应该避免建立和分区列不匹配的索引，除非查询中还同时包含了可以过滤分区的条件。

听起来避免这个问题很简单，不过有时候也会遇到一些意想不到的问题。例如，在一个关联查询中，分区表在关联顺序中是第二个表，并且关联使用的索引和分区条件并不匹配。那么关联时针对第一个表符合条件的每一行，都需要访问并搜索第二个表的所有分区。

注 1： 因为可以在这里存放一个非法的日期，所以甚至当 order_date 是一个非 NULL 值的时候，仍然会出现这样情况。

注 2： 从用户角度来看，这应该是一个缺陷，不过从 MySQL 开发者的角度来看这是一个特性。

选择分区的成本可能很高

如前所述分区有很多类型，不同类型分区的实现方式也不同，所以它们的性能也各不相同。尤其是范围分区，对于回答“这一行属于哪个分区”、“这些符合查询条件的行在哪些分区”这样的问题的成本可能会非常高，因为服务器需要扫描所有的分区定义的列表来找到正确的答案。类似这样的线性搜索的效率不高，所以随着分区数的增长，成本会越来越高。

我们所实际碰到的类似这样的最糟糕的一次问题是按行写入大量数据的时候。每写入一行数据到范围分区的表时，都需要扫描分区定义列表来找到合适的目标分区。可以通过限制分区的数量来缓解此问题，根据实践经验，对大多数系统来说，100 个左右的分区是没有问题的。

其他的分区类型，比如键分区和哈希分区，则没有这样的问题。

打开并锁住所有底层表的成本可能很高

当查询访问分区表的时候，MySQL 需要打开并锁住所有的底层表，这是分区表的另一个开销。这个操作在分区过滤之前发生，所以无法通过分区过滤降低此开销，并且该开销也和分区类型无关，会影响所有的查询。这一点对一些本身操作非常快的查询，比如根据主键查找单行，会带来明显的额外开销。可以用批量操作的方式来降低单个操作的此类开销，例如使用批量插入或者 `LOAD DATA INFILE`、一次删除多行数据，等等。当然同时还是需要限制分区的个数。

维护分区的成本可能很高

某些分区维护操作的速度会非常快，例如新增或者删除分区（当删除一个大分区可能会很慢，不过这是另一回事）。而有些操作，例如重组分区或者类似 `ALTER` 语句的操作：这类操作需要复制数据。重组分区的原理与 `ALTER` 类似，先创建一个临时的分区，然后将数据复制到其中，最后再删除原分区。

如上所述，分区表不是什么“银弹”。下面是目前分区实现中的一些其他限制：

- 所有分区都必须使用相同的存储引擎。
- 分区函数中可以使用的函数和表达式也有一些限制。
- 某些存储引擎不支持分区。 272
- 对于 MyISAM 的分区表，不能再使用 `LOAD INDEX INTO CACHE` 操作。
- 对于 MyISAM 表，使用分区表时需要打开更多的文件描述符。虽然看起来是一个表，其实背后有很多独立的分区，每一个分区对于存储引擎来说都是一个独立的表。这样即使分区表只占用一个表缓存条目，文件描述符还是需要多个。因此，即使已经配置了合适的表缓存，以确保不会超过操作系统的单个进程可以打开的文件描述符的个数，但对于分区表而言，还是会出现超过文件描述符限制的问题。

最后，需要指出的是较老版本的 MySQL 问题会更多些。所有的软件都是有 bug 的。分

区表在 MySQL 5.1 中引入，在后面的 5.1.40 和 5.1.50 之后修复了很多分区表的 bug。在 MySQL 5.5 中，分区表又做了很多改进，这才使得分区表可以逐步考虑用在生产环境了。在即将发布的 MySQL 5.6 版本中，分区表做了更多的增强，例如新引入的 ALTER TABLE EXCHANGE PARTITION。

7.1.5 查询优化

引入分区给查询优化带来了一些新的思路（同时也带来新的 bug）。分区最大的优点就是优化器可以根据分区函数来过滤一些分区。根据粗粒度索引的优势，通过分区过滤通常可以让查询扫描更少的数据（在某些场景下）。

所以，对于访问分区表来说，很重要的一点是要在 WHERE 条件中带入分区列，有时候即使看似多余的也要带上，这样就可以让优化器能够过滤掉无须访问的分区。如果没有这些条件，MySQL 就需要让对应存储引擎访问这个表的所有分区，如果表非常大的话，就可能会非常慢。

使用 EXPLAIN PARTITION 可以观察优化器是否执行了分区过滤，下面是一个示例：

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales_by_day
   partitions: p_2010,p_2011,p_2012
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3
        Extra:
```

正如你所看到的，这个查询将访问所有的分区。下面我们在 WHERE 条件中再加入一个时间限制条件：

273

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day WHERE day > '2011-01-01'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales_by_day
   partitions: p_2011,p_2012
```

MySQL 优化器已经很善于过滤分区。比如它能够将范围条件转化为离散的值列表，并根据列表中的每个值过滤分区。然而，优化器也不是万能的。下面查询的 WHERE 条件理论上可以过滤分区，但实际上却不行：

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day WHERE YEAR(day) = 2010\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales_by_day
   partitions: p_2010,p_2011,p_2012
```

MySQL 只能在使用分区函数的列本身进行比较时才能过滤分区，而不能根据表达式的值去过滤分区，即使这个表达式就是分区函数也不行。这就和查询中使用独立的列才能使用索引的道理是一样的（参考第 5 章的相关内容）。所以只需要把上面的查询等价地改写为如下形式即可：

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day
    -> WHERE day BETWEEN '2010-01-01' AND '2010-12-31'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales_by_day
   partitions: p_2010
```

这里写的 WHERE 条件中带入的是分区列，而不是基于分区列的表达式，所以优化器能够利用这个条件过滤部分分区。一个很重要的原则是：即便在创建分区时可以使用表达式，但在查询时却只能根据列来过滤分区。

优化器在处理查询的过程中总是尽可能聪明地去过滤分区。例如，若分区表是关联操作中的第二张表，且关联条件是分区键，MySQL 就只会在对应的分区里匹配行。（EXPLAIN 无法显示这种情况下的分区过滤，因为这是运行时的分区过滤，而不是查询优化阶段的。）

7.1.6 合并表

合并表（Merge table）是一种早期的、简单的分区实现，和分区表相比有一些不同的限制，并且缺乏优化。分区表严格来说是一个逻辑上的概念，用户无法访问底层的各个分区，对用户来说分区是透明的。但是合并表允许用户单独访问各个子表。分区表和优化器的结合更紧密，这也是未来发展的趋势，而合并表则是一种将被淘汰的技术，在未来的版本中可能被删除。

和分区表类似的是，在 MyISAM 中各个子表可以被一个结构完全相同的逻辑表所封装。可以简单地把这个表当作一个“老的、早期的、功能有限的”的分区表，因为它自身的特性， 274
甚至可以提供一些分区表没有的功能[注3]。

合并表相当于一个容器，里面包含了多个真实表。可以在 CREATE TABLE 中使用一种特别

注 3：这些特性也是一些“鲜为人知的犀利”特性。

的 UNION 语法来指定包含哪些真实表。下面是一个创建合并表的例子：

```
mysql> CREATE TABLE t1(a INT NOT NULL PRIMARY KEY)ENGINE=MyISAM;
mysql> CREATE TABLE t2(a INT NOT NULL PRIMARY KEY)ENGINE=MyISAM;
mysql> INSERT INTO t1(a) VALUES(1),(2);
mysql> INSERT INTO t2(a) VALUES(1),(2);
mysql> CREATE TABLE mrg(a INT NOT NULL PRIMARY KEY)
    -> ENGINE=MERGE UNION=(t1, t2) INSERT_METHOD=LAST;
mysql> SELECT a FROM mrg;
+------+
| a    |
+------+
|    1 |
|    1 |
|    2 |
|    2 |
+------+
```

注意到，这里最后建立的合并表和前面的各个真实表字段完全相同，在合并表中有的索引各个真实子表也有，这是创建合并表的前提条件。另外还注意到，各个子表在对应列上都有主键限制，但是最终的合并表中仍然出现了重复值，这是合并表的另一个不足：合并表中的每一个子表行为和表定义都是相同，但是合并表在全局上并不受这些条件限制。

这里的语法 INSERT_METHOD=LAST 告诉 MySQL，将所有的 INSERT 语句都发送给最后一个表。指定 FIRST 或者 LAST 关键字是唯一可以控制行插入到合并表的哪一个子表的方式（当然，还是可以直接在 SQL 中明确地操作任何一个子表）。而分区表则有更多的方式可以控制数据写入到哪一个子表中。

INSERT 语句的执行结果可以在最终的合并表中看到，也可以在对应的子表中看到：

```
mysql> INSERT INTO mrg(a) VALUES(3);
mysql> SELECT a FROM t2;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
+---+
```

合并表还有些有趣的限制和特性，例如，在删除合并表或者删除一个子表的时候会怎样？
275 删除一个合并表，它的子表不会受任何影响，而如果直接删除其中一个子表则可能会有不同的后果，这要视操作系统而定。例如在 GNU/Linux 上，如果子表的文件描述还是被打开的状态，那么这个表还存在，但是只能通过合并表才能访问到：

```
mysql> DROP TABLE t1, t2;
mysql> SELECT a FROM mrg;
+------+
| a    |
+------+
|    1 |
|    1 |
|    2 |
|    2 |
|    3 |
+------+
```

合并表还有很多其他的限制和行为，下面列举的这几点需要在使用的时候时刻记住。

- 在使用 CREATE 语句创建一个合并表的时候，并不会检查各个子表的兼容性。如果子表的定义稍有不同，那么 MySQL 就可能创建出一个后面无法使用的合并表。另外，如果在成功创建了合并表后再修改某个子表的定义，那么之后再使用合并表可能会看到这样的报错：ERROR 1168 (HY000): Unable to open underlying table which is differently defined or of non-MyISAM type or doesn't exist。
- 根据合并表的特性，不难发现，在合并表上无法使用 REPLACE 语法，无法使用自增字段。更多的细节请参阅 MySQL 官方手册。
- 如果一个查询访问合并表，那么它需要访问所有子表。这会让根据键查找单行的查询速度变慢，如果能够只访问一个对应表，速度肯定将更快。所以，限制合并表中的子表数量很重要，特别是当合并表是某个关联查询的一部分的时候，因为这时访问一个表的记录数可能会将比较操作传递到关联的其他表中，这时减少记录的访问就是减少整个关联操作。当你打算使用合并表的时候，还需要记住以下几点：
    - 执行范围查询时，需要在每一个子表上各执行一次，这比直接访问单个表的性能要差很多，而且子表越多，性能越糟。
    - 全表扫描和普通表的全表扫描速度相同。
    - 在合并表上做唯一键和主键查询时，一旦找到一行数据就会停止。所以一旦查询在合并表的某一个子表中找到一行数据，就会立刻返回，不会再访问任何其他的表。
    - 子表的读取顺序和 CREATE TABLE 语句中的顺序相同。如果需要频繁地按照某个特定顺序访问表，那么可以通过这个特性来让合并排序操作更高效。

因为合并表的各个子表可以直接被访问，所以它还具有一些 MySQL 5.5 分区所不能提供的特性： 276

- 一个 MyISAM 表可以是多个合并表的子表。
- 可以通过直接复制 *.frm*、*.MYI*、*.MYD* 文件，来实现在不同的服务器之间复制各个子表。

- 在合并表中可以很容易地添加新的子表：直接修改合并表的定义就可以了。
- 可以创建一个合并表，让它只包含需要的数据，例如只包含某个时间段的数据，而在分区表中是做不到这一点的。
- 如果想对某个子表做备份、恢复、修改、修复或者别的操作时，可以先将其从合并表中删除，操作结束后再将其加回去。
- 可以使用 *myisampack* 来压缩所有的子表。

相反，分区表的子表都是被 MySQL 隐藏的，只能通过分区表去访问子表。

7.2 视图

MySQL 5.0 版本之后开始引入视图。视图本身是一个虚拟表，不存放任何数据。在使用 SQL 语句访问视图的时候，它返回的数据是 MySQL 从其他表中生成的。视图和表是在同一个命名空间，MySQL 在很多地方对于视图和表是同样对待的。不过视图和表也有不同，例如，不能对视图创建触发器，也不能使用 DROP TABLE 命令删除视图。

在 MySQL 官方手册中对如何创建和使用视图有详细的介绍，本书不会详细介绍这些。我们将主要介绍视图是如何实现的，以及优化器如何处理视图，通过了解这些，希望可以让大家在使用视图时获得更高的性能。我们将使用示例数据库 world 来演示视图是如何工作的：

```
mysql> CREATE VIEW Oceania AS
    ->    SELECT * FROM Country WHERE Continent = 'Oceania'
    ->    WITH CHECK OPTION;
```

实现视图最简单的方法是将 SELECT 语句的结果存放到临时表中。当需要访问视图的时候，直接访问这个临时表就可以了。我们先来看看下面的查询：

```
mysql> SELECT Code, Name FROM Oceania WHERE Name = 'Australia';
```

下面是使用临时表来模拟视图的方法。这里临时表的名字是为演示用的：

```
mysql> CREATE TEMPORARY TABLE TMP_Oceania_123 AS
    ->    SELECT * FROM Country WHERE Continent = 'Oceania';
mysql> SELECT Code, Name FROM TMP_Oceania_123 WHERE Name = 'Australia';
```

277 这样做会有明显的性能问题，优化器也很难优化在这个临时表上的查询。实现视图更好的方法是，重写含有视图的查询，将视图的定义 SQL 直接包含进查询的 SQL 中。下面的例子展示的是将视图定义的 SQL 合并进查询 SQL 后的样子：

```
mysql> SELECT Code, Name FROM Country
    -> WHERE Continent = 'Oceania' AND Name = 'Australia';
```

MySQL 可以使用这两种办法中的任何一种来处理视图。这两种算法分别称为合并算法（MERGE）和临时表算法（TEMPTABLE）注4，如果可能，会尽可能地使用合并算法。MySQL 甚至可以嵌套地定义视图，也就是在一个视图上再定义另一个视图。可以在 EXPLAIN EXTENDED 之后使用 SHOW WARNINGS 来查看使用视图的查询重写后的结果。

如果是采用临时表算法实现的视图，EXPLAIN 中会显示为派生表（DERIVED）。图 7-1 展示了这两种实现的细节。

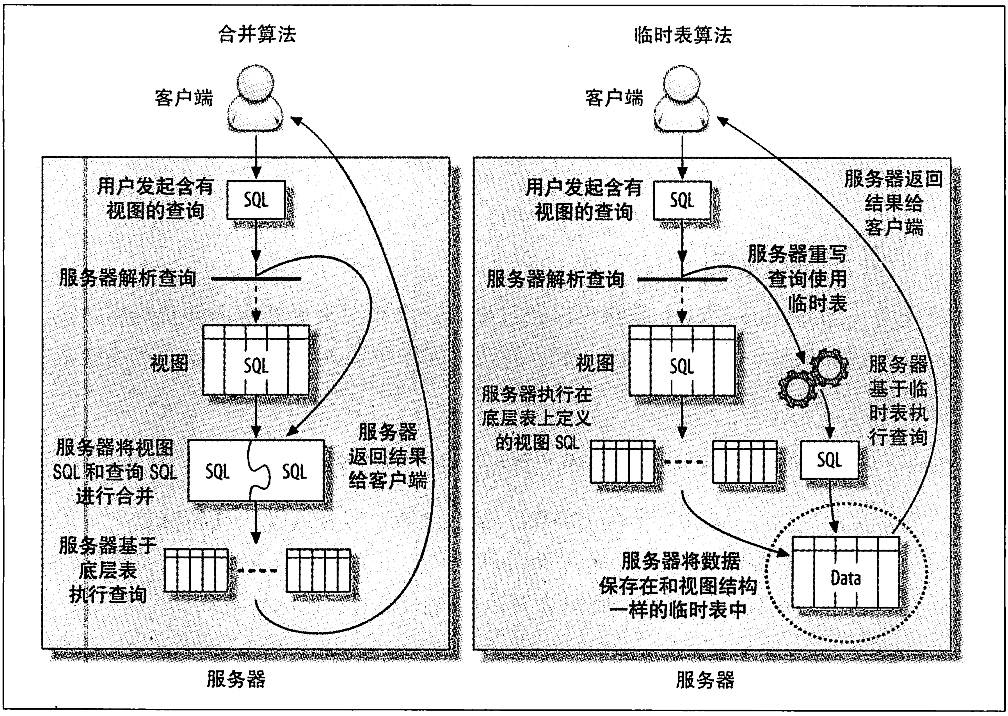


图7-1：视图的两种实现

如果视图中包含 GROUY BY、DISTINCT、任何聚合函数、UNION、子查询等，只要无法在原表记录和视图记录中建立一一映射的场景中，MySQL 都将使用临时表算法来实现视图。上面列举的可能不全，而且这些规则在未来的版本中也可能会改变。如果你想确定 278
MySQL 到底是使用合并算法还是临时表算法，可以 EXPLAIN 一条针对视图的简单查询：

注 4： 这里的“temp table”并不是指真正的物理上存在的临时表。没有经过这些改进和试验，MySQL 视图也不会有现在的效率。

```
mysql> EXPLAIN SELECT * FROM <view_name>;
+----+-------------+
| id | select_type |
+----+-------------+
|  1 | PRIMARY     |
|  2 | DERIVED     |
+----+-------------+
```

这里的 select_type 为“DERIVED”，说明该视图是采用临时表算法实现的。不过要注意：如果产生的底层派生表很大，那么执行 EXPLAIN 可能会非常慢。因为在 MySQL 5.5 和更老的版本中，EXPLAIN 是需要实际执行并产生该派生表的。

视图的实现算法是视图本身的属性，和作用在视图上的查询语句无关。例如，可以为一个基于简单查询的视图指定使用临时表算法：

```
CREATE ALGORITHM=TEMPTABLE VIEW v1 AS SELECT * FROM sakila.actor;
```

实现该视图的 SQL 本身并不需要临时表，但基于该视图无论执行什么样的查询，视图都会生成一个临时表。

7.2.1 可更新视图

可更新视图（updatable view）是指可以通过更新这个视图来更新视图涉及的相关表。只要指定了合适的条件，就可以更新、删除甚至向视图中写入数据。例如，下面就是一个合理的操作：

```
mysql> UPDATE Oceania SET Population = Population * 1.1 WHERE Name = 'Australia';
```

如果视图定义中包含了 GROUP BY、UNION、聚合函数，以及其他一些特殊情况，就不能被更新了。更新视图的查询也可以是一个关联语句，但是有一个限制，被更新的列必须来自同一个表中。另外，所有使用临时表算法实现的视图都无法被更新。

在上一节定义视图时使用的 CHECK OPTION 子句，表示任何通过视图更新的行，都必须符合视图本身的 WHERE 条件定义。所以不能更新视图定义列以外的列，比如上例中不能更新 Continent 列，也不能插入不同 Continent 值的新数据，否则 MySQL 会报如下的错误：

```
mysql> UPDATE Oceania SET Continent = 'Atlantis';
ERROR 1369 (HY000): CHECK OPTION failed 'world.Oceania'
```

某些关系数据库允许在视图上建立 INSTEAD OF 触发器，通过触发器可以精确控制在修改视图数据时做些什么。不过 MySQL 不支持在视图上建任何触发器。

7.2.2 视图对性能的影响

多数人认为视图不能提升性能，实际上，在 MySQL 中某些情况下视图也可以帮助提升性能。而且视图还可以和其他提升性能的方式叠加使用。例如，在重构 schema 的时候可以使用视图，使得在修改视图底层表结构的时候，应用代码还可能继续不报错的运行。

可以使用视图实现基于列的权限控制，却不需要真正的在系统中创建列权限，因此没有额外的开销。

```
CREATE VIEW public.employeeinfo AS
   SELECT firstname, lastname -- but not socialsecuritynumber
   FROM private.employeeinfo;
GRANT SELECT ON public.* TO public_user;
```

有时候也可以使用伪临时视图实现一些功能。MySQL 虽然不能创建只在当前连接中存在的真正的临时视图，但是可以建一个特殊名字的视图，然后在连接结束的时候删除该视图。这样在连接过程中就可以在 FROM 子句中使用这个视图，和使用子查询的方式完全相同，因为 MySQL 在处理视图和处理子查询的代码路径完全不同，所以它们的性能也不同。下面是一个例子：

```
-- Assuming 1234 is the result of CONNECTION_ID()
CREATE VIEW temp.cost_per_day_1234 AS
   SELECT DATE(ts) AS day, sum(cost) AS cost
   FROM logs.cost
   GROUP BY day;
SELECT c.day, c.cost, s.sales
FROM temp.cost_per_day_1234 AS c
   INNER JOIN sales.sales_per_day AS s USING(day);
DROP VIEW temp.cost_per_day_1234;
```

我们这里使用连接 ID 作为视图名字的一部分来避免冲突。在应用发生崩溃和别的意外导致未清理临时视图的时候，这个技巧使得清理临时视图变得很简单。详细的信息可以参考后面的“丢失的临时表”。

使用临时表算法实现的视图，在某些时候性能会很糟糕（虽然可能比直接使用等效查询语句要好一点）。MySQL 以递归的方式执行这类视图，先会执行外层查询，即使外层查询优化器将其优化得很好，但是 MySQL 优化器可能无法像其他的数据库那样做更多的内外结合的优化。外层查询的 WHERE 条件无法“下推”到构建视图的临时表的查询中，临时表也无法建立索引[注5]。下面是一个例子，还是基于 temp.cost_per_day_1234 这个视图：

注 5： 在 MySQL 5.6 中可能会有所改进，但是在本书写作的时候 5.6 还没有发布。

280

```
mysql> SELECT c.day, c.cost, s.sales
    -> FROM temp.cost_per_day_1234 AS c
    ->    INNER JOIN sales.sales_per_day AS s USING(day)
    ->    WHERE day BETWEEN '2007-01-01' AND '2007-01-31';
```

在这个查询中，MySQL 先执行视图的 SQL 生成临时表，然后再将 sales_per_day 和临时表进行关联。这里的 WHERE 子句中的 BETWEEN 条件并不能下推到视图当中，所以视图在创建的时候仍然需要将所有的数据都放到临时表当中，而不仅仅是一个月的数据。而且临时表中不会有索引。这个案例中，索引还不是问题：MySQL 将临时表作为关联顺序中的第一个表，因此这里可以使用 sales_per_day 中的索引。不过，如果是对两个视图做关联的话，优化器就没有任何索引可以使用了。

视图还引入了一些并非 MySQL 特有的其他问题。很多开发者以为视图很简单，但实际上其背后的逻辑可能非常复杂。开发人员如果没有意识到视图背后的复杂性，很可能会以为是在不停地重复查询一张简单的表，而没有意识到实际上是代价高昂的视图。我们见过不少案例，一条看起来很简单的查询，EXPLAIN 出来却有几百行，因为其中一个或者多个表，实际上是引用了很多其他表的视图。

如果打算使用视图来提升性能，需要做比较详细的测试。即使是合并算法实现的视图也会有额外的开销，而且视图的性能很难预测。在 MySQL 优化器中，视图的代码执行路径也完全不同，这部分代码测试还不够全面，可能会有一些隐藏缺陷和问题。所以，我们认为视图还不是那么成熟。例如，我们看到过这样的案例，复杂的视图和高并发的查询导致查询优化器花了大量时间在执行计划生成和统计数据阶段，这甚至会导致 MySQL 服务器僵死，后来通过将视图转换成等价的查询语句解决了问题。这也说明视图——即使是使用合并算法实现的——并不总是有很优化的实现。

7.2.3 视图的限制

在其他的关系数据库中你可能使用过物化视图，MySQL 还不支持物化视图（物化视图是指将视图结果数据存放在一个可以查看的表中，并定期从原始表中刷新数据到这个表中）。MySQL 也不支持在视图中创建索引。不过，可以使用构建缓存表或者汇总表的办法来模拟物化视图和索引。可以直接使用 Justin Swanhart's 的工具 Flexviews 来实现这个目的。参考第 4 章可以获得更多的相关细节。

MySQL 视图实现上也有一些让人烦恼的地方。例如，MySQL 并不会保存视图定义的原始 SQL 语句，所以如果打算通过执行 SHOW CREATE VIEW 后再简单地修改其结果的方式来重新定义视图，可能会大失所望。SHOW CREATE VIEW 出来的视图创建语句将以一种不
281 友好的内部格式呈现，充满了各种转义符和引号，没有代码格式化，没有注释，也没有缩进。

如果打算重新修改一个视图，并且没法找到视图的原始的创建语句的话，可以通过使用视图的 *.frm* 文件的最后一行获得一些信息。如果有 FILE 权限，甚至可以直接使用 SQL 语句中的 LOAD_FILE() 来读取 *.frm* 中的视图创建信息。再加上一些字符处理工作，就可以获得一个完整的视图创建语句了，感谢 Roland Bouman 创造性的实现：

```
mysql> SELECT
    ->    REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(
    ->    REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(
    ->       SUBSTRING_INDEX(LOAD_FILE('/var/lib/mysql/world/Oceania.frm'),
    ->       '\nsource=', -1),
    ->    '\\_','\_'), '\\%','\%'), '\\\\','\\'), '\\Z','\Z'), '\\t','\t'),
    ->    '\\r','\r'), '\\n','\n'), '\\b','\b'), '\\\"','\"'), '\\\'','\''),
    ->    '\\0','\0')
    -> AS source;
+-------------------------------------------------------------------+
| source                                                            |
+-------------------------------------------------------------------+
| SELECT * FROM Country WHERE continent = 'Oceania'
  WITH CHECK OPTION
|
+-------------------------------------------------------------------+
```

7.3 外键约束

InnoDB 是目前 MySQL 中唯一支持外键的内置存储引擎，所以如果需要外键支持那选择就不多了（PBXT 也有外键支持）。

使用外键是有成本的。比如外键通常都要求每次在修改数据时都要在另外一张表中多执行一次查找操作。虽然 InnoDB 强制外键使用索引，但还是无法消除这种约束检查的开销。如果外键列的选择性很低，则会导致一个非常大且选择性很低的索引。例如，在一个非常大的表上有 status 列，并希望限制这个状态列的取值，如果该列只能取三个值——虽然这个列本身很小，但是如果主键很大，那么这个索引就会很大——而且这个索引除了做这个外键限制，也没有任何其他的作用了。

不过，在某些场景下，外键会提升一些性能。如果想确保两个相关表始终有一致的数据，那么使用外键比在应用程序中检查一致性的性能要高得多，此外，外键在相关数据的删除和更新上，也比在应用中维护要更高效，不过，外键维护操作是逐行进行的，所以这样的更新会比批量删除和更新要慢些。

外键约束使得查询需要额外访问一些别的表，这也意味着需要额外的锁。如果向子表中写入一条记录，外键约束会让 InnoDB 检查对应的父表的记录，也就需要对父表对应记录进行加锁操作，来确保这条记录不会在这个事务完成之时就被删除了。这会导致额外的锁等待，甚至会导致一些死锁。因为没有直接访问这些表，所以这类死锁问题往往难以排查。 282

有时，可以使用触发器来代替外键。对于相关数据的同时更新外键更合适，但是如果外键只是用作数值约束，那么触发器或者显式地限制取值会更好些。（这里，可以直接使用 ENUM 类型。）

如果只是使用外键做约束，那通常在应用程序里实现该约束会更好。外键会带来很大的额外消耗。这里没有相关的基准测试的数据，不过我们碰到过很多案例，在对性能进行剖析时发现外键约束就是瓶颈所在，删除外键后性能立即大幅提升。

7.4 在 MySQL 内部存储代码

MySQL 允许通过触发器、存储过程、函数的形式来存储代码。从 MySQL 5.1 开始，还可以在定时任务中存放代码，这个定时任务也被称为“事件”。存储过程和存储函数都被统称为“存储程序”。

这四种存储代码都使用特殊的 SQL 语句扩展，它包含了很多过程处理语法，例如循环和条件分支等[注6]。不同类型的存储代码的主要区别在于其执行的上下文——也就是其输入和输出。存储过程和存储函数都可以接收参数然后返回值，但是触发器和事件却不行。

一般来说，存储代码是一种很好的共享和复用代码的方法。Giuseppe Maxia 和其他一些人也建立了一些通用的存储过程库，在网站 *http://mysql-sr-lib.sourceforge.net* 可以找到。不过因为不同的关系数据库都有各自的语法规则，所以不同的数据库很难复用这些存储代码（DB2 是一个例外，它和 MySQL 基于相同的标准，有着非常类似的语法）[注7]。

这里将主要关注存储代码的性能，而不是如何实现。如果你打算学习如何编写存储过程，那么 Guy Harrison 和 Steven Feuerstein 编写的 *MySQL Stored Procedure Programming*（O'Reilly）应该会有帮助。

有人倡导使用存储代码，也有人反对。这里我们不站在任何一边，只是列举一下在 MySQL 中使用存储代码的优点和缺点。首先，它有如下优点：

283
- 它在服务器内部执行，离数据最近，另外在服务器上执行还可以节省带宽和网络延迟。
- 这是一种代码重用。可以方便地统一业务规则，保证某些行为总是一致，所以也可以为应用提供一定的安全性。
- 它可以简化代码的维护和版本更新。

注 6：这个语法是 SQL/PSM 的一个子集，SQL/PSM 是 SQL 标准中的持久化存储模块，在 ISO/IEC 9075-4:2003(E) 中定义。

注 7：有一些专门用作移植的工具，例如 *tsql2mysql* 项目就是专门用于移植 SQL Server 上的存储过程。参考：*http://sourceforge.net/projects/tsql2mysql*。

- 它可以帮助提升安全，比如提供更细粒度的权限控制。一个常见的例子是银行用于转移资金的存储过程：这个存储过程可以在一个事务中完成资金转移和记录用于审计的日志。应用程序也可以通过存储过程的接口访问那些没有权限的表。
- 服务器端可以缓存存储过程的执行计划，这对于需要反复调用的过程，会大大降低消耗。
- 因为是在服务器端部署的，所以备份、维护都可以在服务器端完成。所以存储程序的维护工作会很简单。它没什么外部依赖，例如，不依赖任何 Perl 包和其他不想在服务器上部署的外部软件。
- 它可以在应用开发和数据库开发人员之间更好地分工。不过最好是由数据库专家来开发存储过程，因为不是每个应用开发人员都能写出高效的 SQL 查询。

存储代码也有如下缺点：

- MySQL 本身没有提供好用的开发和调试工具，所以编写 MySQL 的存储代码比其他的数据库要更难些。
- 较之应用程序的代码，存储代码效率要稍微差些。例如，存储代码中可以使用的函数非常有限，所以使用存储代码很难编写复杂的字符串维护功能，也很难实现太复杂的逻辑。
- 存储代码可能会给应用程序代码的部署带来额外的复杂性。原本只需要部署应用代码和库表结构变更，现在还需要额外地部署 MySQL 内部的存储代码。
- 因为存储程序都部署在服务器内，所以可能有安全隐患。如果将非标准的加密功能放在存储程序中，那么若数据库被攻破，数据也就泄漏了。但是若将加密函数放在应用程序代码中，那么攻击者必须同时攻破程序和数据库才能获得数据。
- 存储过程会给数据库服务器增加额外的压力，而数据库服务器的扩展性相比应用服务器要差很多。
- MySQL 并没有什么选项可以控制存储程序的资源消耗，所以在存储过程中的一个小错误，可能直接把服务器拖死。
- 存储代码在 MySQL 中的实现也有很多限制——执行计划缓存是连接级别的，游标的物化和临时表相同，在 MySQL 5.5 版本之前，异常处理也非常困难，等等。（我 284
们会在介绍它的各个特性的同时介绍相关的限制）。简而言之，较之 T-SQL 或者 PL/SQL，MySQL 的存储代码功能还非常非常弱。
- 调试 MySQL 的存储过程是一件很困难的事情。如果慢日志只是给出 `CALL XYZ('A')`，通常很难定位到底是什么导致的问题，这时不得不看看存储过程中的 SQL 语句是如何编写的。（这在 Percona Server 中可以通过参数控制。）

- 它和基于语句的二进制日志复制合作得并不好。在基于语句的复制中，使用存储代码通常有很多的陷阱，除非你在这方面的经验非常丰富或者非常有耐心排查这类问题，否则需要谨慎使用。

这个缺陷列表很长——那么在真实世界中，这意味着什么？我们来看一个真实世界中弄巧成拙的案例：在一个实例中，创建了一个存储过程来给应用程序访问数据库中的数据，这使得所有的数据访问都需要通过这个接口，甚至很多根据主键的查询也是如此，这大概使系统的性能降低了五倍左右。

最后，存储代码是一种帮助应用隐藏复杂性，使得应用开发更简单的方法。不过，它的性能可能更低，而且会给 MySQL 的复制等增加潜在的风险。所以当你打算使用存储过程的时候，需要问问自己，到底希望程序逻辑在哪儿实现：是数据库中还是应用代码中？这两种做法都可以，也都很流行。只是当你编写存储代码的时候，你需要明白这是将程序逻辑放在数据库中。

7.4.1 存储过程和函数

MySQL 的架构本身和优化器的特性使得存储代码有一些天然的限制，它的性能也一定程度受限于此。在本书编写的时候，有如下的限制：

- 优化器无法使用关键字 `DETERMINISTIC` 来优化单个查询中多次调用存储函数的情况。
- 优化器无法评估存储函数的执行成本。
- 每个连接都有独立的存储过程的执行计划缓存。如果有多个连接需要调用同一个存储过程，将会浪费缓存空间来反复缓存同样的执行计划。（如果使用的是连接池或者是持久化连接，那么执行计划缓存可能会有更长的生命周期。）
- 存储程序和复制是一组诡异组合。如果可以，最好不要复制对存储程序的调用。直接复制由存储程序改变的数据则会更好。MySQL 5.1 引入的行复制能够改善这个问题。如果在 MySQL 5.0 中开启了二进制日志，那么要么在所有的存储过程中都增加 `DETERMINISTIC` 限制或者设置 MySQL 的选项 `log_bin_trust_function_creators`。

285 我们通常会希望存储程序越小、越简单越好。希望将更加复杂的处理逻辑交给上层的应用实现，通常这样会使代码更易读、易维护，也会更灵活。这样做也会让你拥有更多的计算资源，潜在的还会让你拥有更多的缓存资源[注8]。

不过，对于某些操作，存储过程比其他的实现要快得多——特别是当一个存储过程调用可以代替很多小查询的时候。如果查询很小，相比这个查询执行的成本，解析和网络开销就变得非常明显。为了证明这一点，我们先创建一个简单的存储过程，用来写入一定

注 8： 通常各个层都有自己的缓存。——译者注

数量的数据到一个表中，下面是存储过程的代码：

```
DROP PROCEDURE IF EXISTS insert_many_rows;

delimiter //

CREATE PROCEDURE insert_many_rows (IN loops INT)
BEGIN
   DECLARE v1 INT;
   SET v1=loops;
   WHILE v1 > 0 DO
     INSERT INTO test_table values(NULL,0,
               'qqqqqqqqqqwwwwwwwwwweeeeeeeeeerrrrrrrrrrtttttttttt',
               'qqqqqqqqqqwwwwwwwwwweeeeeeeeeerrrrrrrrrrtttttttttt');
     SET v1 = v1 - 1;
   END WHILE;
END;
//

delimiter ;
```

然后对该存储过程执行基准测试，看插入一百万条记录的时间，并和通过客户端程序逐条插入一百万条记录的时间进行对比。这里表结构和硬件并不重要——重要的是两种方式的相对速度。另外，我们还测试了使用 MySQL Proxy 连接 MySQL 来执行客户端程序测试的性能。为了让事情简单，整个测试在一台服务器上完成，包括客户端程序和 MySQL Proxy 实例。表 7-1 展示了测试结果。

表7-1：写入一百万数据所花费的总时间

写入方式	总消耗时间
存储过程	101 sec
客户端程序	279 sec
使用 MySQL Proxy 的客户端程序	307 sec

可以看到存储过程要快很多，很大程度因为它无须网络通信开销、解析开销和优化器开销等。

我们将在本章的后半部分介绍如何维护存储过程。

7.4.2 触发器 286

触发器可以让你在执行 INSERT、UPDATE 或者 DELETE 的时候，执行一些特定的操作。可以在 MySQL 中指定是在 SQL 语句执行前触发还是在执行后触发。触发器本身没有返回值，不过它们可以读取或者改变触发 SQL 语句所影响的数据。所以，可以使用触发器实现一些强制限制，或者某些业务逻辑，否则，就需要在应用程序中实现这些逻辑。

因为使用触发器可以减少客户端和服务器之间的通信，所以触发器可以简化应用逻辑，还可以提高性能。另外，还可以用于自动更新反范式化数据或者汇总表数据。例如，在示例数据库 Sakila 中，我们可以使用触发器来维护 film_text 表。

MySQL 触发器的实现非常简单，所以功能也有限。如果你在其他数据库产品中已经重度依赖触发器，那么在使用 MySQL 的时候需要注意，很多时候 MySQL 触发器的表现和预想的并不一样。特别需要注意以下几点：

- 对每一个表的每一个事件，最多只能定义一个触发器（换句话说，不能在 AFTER INSERT 上定义两个触发器）。
- MySQL 只支持“基于行的触发”——也就是说，触发器始终是针对一条记录的，而不是针对整个 SQL 语句的。如果变更的数据集非常大的话，效率会很低。

下面这些触发器本身的限制也适用于 MySQL：

- 触发器可以掩盖服务器背后的工作，一个简单的 SQL 语句背后，因为触发器，可能包含了很多看不见的工作。例如，触发器可能会更新另一个相关表，那么这个触发器会让这条 SQL 影响的记录数翻一倍。
- 触发器的问题也很难排查，如果某个性能问题和触发器相关，会很难分析和定位。
- 触发器可能导致死锁和锁等待。如果触发器失败，那么原来的 SQL 语句也会失败。如果没有意识到这其中是触发器在搞鬼，那么很难理解服务器抛出的错误代码是什么意思。

如果仅考虑性能，那么 MySQL 触发器的实现中对服务器限制最大的就是它的“基于行的触发”设计。因为性能的原因，很多时候无法使用触发器来维护汇总和缓存表。使用触发器而不是批量更新的一个重要原因就是，使用触发器可以保证数据总是一致的。

触发器并不能一定保证更新的原子性。例如，一个触发器在更新 MyISAM 表的时候，如果遇到什么错误，是没有办法做回滚操作的。这时，触发器可以抛出错误。假设你在一个 MyISAM 表上建立一个 AFTER UPDATE 的触发器，用来更新另一个 MyISAM 表。如果触发器在更新第二个表的时候遇到错误导致更新失败，那么第一个表的更新并不会回滚。

287 在 InnoDB 表上的触发器是在同一个事务中完成的，所以它们执行的操作是原子的，原操作和触发器操作会同时失败或者成功。不过，如果在 InnoDB 表上建触发器去检查数据的一致性，需要特别小心 MVCC，稍不小心，你可能会获得错误的结果。假设，你想实现外键约束，但是不打算使用 InnoDB 的外键约束。若打算编写一个 BEFORE INSERT 触发器来检查写入的数据对应列在另一个表中是存在的，但若你在触发器中没有使用 SELECT FOR UPDATE，那么并发的更新语句可能会立刻更新对应记录，导致数据不一致。

我们不是危言耸听，让大家不要使用触发器。相反，触发器非常有用，尤其是实现一些约束、系统维护任务，以及更新反范式化数据的时候。

还可以使用触发器来记录数据变更日志。这对实现一些自定义的复制会非常方便，比如需要先断开连接，然后修改数据，最后再将所有的修改重新合并回去的情况。一个简单的例子是，一组用户各自在自己的个人电脑上工作，但他们的操作都需要同步到一台主数据库上，然后主数据库会将他们所有人的操作都分发给每个人。实现这个系统需要做两次同步操作。触发器就是构建整个系统的一个好办法。每个人的电脑上都可以使用一个触发器来记录每一次数据的修改，并将其发送到主数据库中。然后，再使用 MySQL 的复制将主数据库上的所有操作都复制一份到本地并应用。这里需要额外注意的是，如果触发器基于有自增主键的记录，并且使用的是基于语句的复制，那么自增长可能会在复制中出现不一致。

有时候可以使用一些技巧绕过触发器是“基于行的触发”这个限制。Roland Bouman 发现，对于 BEFORE 触发器除了处理的第一条记录，触发器函数 ROW_COUNT() 总是会返回 1。可以使用这个特点，使得触发器不再是针对每一行都运行，而是针对一条 SQL 语句运行一次。这和真正意义上的单条 SQL 语句的触发器并不相同，不过可以使用这个技术来模拟单条 SQL 语句的 BEFORE 触发器。这个行为可能是 MySQL 的一个缺陷，未来版本中可能会被修复，所以在使用这个技巧的时候，需要先验证在你的 MySQL 版本中是否适用，另外，在升级数据库的时候还需要检查这类触发器是否还能够正常工作。下面是一个使用这个技巧的例子：

```
CREATE TRIGGER fake_statement_trigger
BEFORE INSERT ON sometable
FOR EACH ROW
BEGIN
   DECLARE v_row_count INT DEFAULT ROW_COUNT();
   IF v_row_count <> 1 THEN
      -- Your code here
   END IF;
END;
```

7.4.3 事件 288

事件是 MySQL 5.1 引入的一种新的存储代码的方式。它类似于 Linux 的定时任务，不过是完全在 MySQL 内部实现的。你可以创建事件，指定 MySQL 在某个时候执行一段 SQL 代码，或者每隔一个时间间隔执行一段 SQL 代码。通常，我们会把复杂的 SQL 都封装到一个存储过程中，这样事件在执行的时候只需要做一个简单的 CALL 调用。

事件在一个独立事件调度线程中被初始化，这个线程和处理连接的线程没有任何关系。它不接收任何参数，也没有任何的返回值。可以在 MySQL 的日志中看到命令的执行日志，

还可以在表 INFORMATION_SCHEMA.EVENTS 中看到各个事件状态，例如这个事件最后一次被执行的时间等。

类似的，一些适用于存储过程的考虑也同样适用于事件。首先，创建事件意味着给服务器带来额外工作。事件实现机制本身的开销并不大，但是事件需要执行 SQL，则可能会对性能有很大的影响。更进一步，事件和其他的存储程序一样，在和基于语句的复制一起工作时，也可能会触发同样的问题。事件的一些典型应用包括定期地维护任务、重建缓存、构建汇总表来模拟物化视图，或者存储用于监控和诊断的状态值。

下面的例子创建了一个事件，它会每周一次针对某个数据库运行一个存储过程（后面我们将展示如何创建这个存储过程）：

```
CREATE EVENT optimize_somedb ON SCHEDULE EVERY 1 WEEK
DO
CALL optimize_tables('somedb');
```

你可以指定事件本身是否被复制。根据需要，有时需要被复制，有时则不需要。看前面的例子，你可能会希望在所有的备库上都运行 OPTIMIZE TABLE，不过要注意如果所有的备库同时执行，可能会影响服务器的性能（会对表加锁）。

最后，如果一个定时事件执行需要很长的时间，那么有可能会出现这样的情况，即前面一个事件还未执行完成，下一个时间点的事件又开始了。MySQL 本身不会防止这种并发，所以需要用户自己编写这种情况下的防并发代码。你可以使用函数 GET_LOCK() 来确保当前总是只有一个事件在被执行：

```
CREATE EVENT optimize_somedb ON SCHEDULE EVERY 1 WEEK
DO
BEGIN
   DECLARE CONTINUE HANLDER FOR SQLEXCEPTION
      BEGIN END;
   IF GET_LOCK('somedb', 0) THEN
      DO CALL optimize_tables('somedb');
   END IF;
   DO RELEASE_LOCK('somedb');
END
```

289 这里的"CONTINUE HANLDER"用来确保，即使当事件执行出现了异样，仍然会释放持有的锁。

虽然事件的执行是和连接无关的，但是它仍然是线程级别的。MySQL 中有一个事件调度线程，必须在 MySQL 配置文件中设置，或者使用下面的命令来设置：

```
mysql> SET GLOBAL event_scheduler := 1;
```

该选项一旦设置，该线程就会执行各个用户指定的事件中的各段 SQL 代码。你可以通过观察 MySQL 的错误日志来了解事件的执行情况。

虽然事件调度是一个单独的线程，但是事件本身是可以并行执行的。MySQL 会创建一个新的进程用于事件执行。在事件的代码中，如果你调用函数 CONNECTION_ID()，也会返回一个唯一值，和一般的线程返回值一样——虽然事件和 MySQL 的连接线程是无关的（这里的函数 CONNECTION_ID() 返回的只是线程 ID）。这里的进程和线程生命周期就是事件的执行过程。可以通过 SHOW PROCESSLIST 中的 Command 列来查看，这些线程的该列总是显示为“Connect”。

虽然事件处理进程需要创建一个线程来真正地执行事件，但该线程在时间执行结束后会被销毁，而不会放到线程缓存中，并且状态值 Threads_created 也不会被增加。

7.4.4 在存储程序中保留注释

存储过程、存储函数、触发器、事件通常都会包含大量的重要代码，在这些代码中加上注释就非常有必要了。但是这些注释可能不会存储在 MySQL 服务器中，因为 MySQL 的命令行客户端会自动过滤注释（命令行客户端的这个“特性”令人生厌，不过这就是生活）。

一个将注释存储到存储程序中的技巧就是使用版本相关的注释，因为这样的注释可能被 MySQL 服务器执行（例如，只有版本号大于某个值的时候才执行的代码）。服务器和客户端都知道这不是普通的注释，所以也就不会删除这些注释。为了让这样的“版本相关的代码”不被执行，可以指定一个非常大的版本号，例如 99 999。我们现在给触发器加上一些注释文档，让它更易读：

```
CREATE TRIGGER fake_statement_trigger
BEFORE INSERT ON sometable
FOR EACH ROW
BEGIN
   DECLARE v_row_count INT DEFAULT ROW_COUNT();
   /*!99999     ROW_COUNT() is 1 except for the first row, so this executes
      only once per statement.   */
   IF v_row_count <> 1 THEN
      -- Your code here
   END IF;
END;
```

7.5 游标

290

MySQL 在服务器端提供只读的、单向的游标，而且只能在存储过程或者更底层的客户端 API 中使用。因为 MySQL 游标中指向的对象都是存储在临时表中而不是实际查询到的数据，所以 MySQL 游标总是只读的。它可以逐行指向查询结果，然后让程序做进一步的处理。在一个存储过程中，可以有多个游标，也可以在循环中“嵌套”地使用游标。

MySQL 的游标设计也为粗心的人“准备”了陷阱。因为是使用临时表实现的，所以它在效率上给开发人员一个错觉。需要记住的最重要的一点是：当你打开一个游标的时候需要执行整个查询。考虑下面的存储过程：

```
1 CREATE PROCEDURE bad_cursor()
2 BEGIN
3    DECLARE film_id INT;
4    DECLARE f CURSOR FOR SELECT film_id FROM sakila.film;
5    OPEN f;
6    FETCH f INTO film_id;
7    CLOSE f;
8 END
```

从这个例子中可以看到，不用处理完所有的数据就可以立刻关闭游标。使用 Oracle 或者 SQL Server 的用户不会认为这个存储过程有什么问题，但是在 MySQL 中，这会带来很多的不必要的额外操作。使用 `SHOW STATUS` 来诊断这个存储过程，可以看到它需要做 1 000 个索引页的读取，做 1 000 个写入。这是因为在表 `sakila.film` 中有 1 000 条记录，而所有这些读和写都发生在第五行的打开游标动作。

这个案例告诉我们，如果在关闭游标的时候你只是扫描一个大结果集的一小部分，那么存储过程可能不仅没有减少开销，相反带来了大量的额外开销。这时，你需要考虑使用 `LIMIT` 来限制返回的结果集。

游标也会让 MySQL 执行一些额外的 I/O 操作，而这些操作的效率可能非常低。因为临时内存表不支持 `BLOB` 和 `TEXT` 类型，如果游标返回的结果包含这样的列的话，MySQL 就必须创建临时磁盘表来存放，这样性能可能会很糟。即使没有这样的列，当临时表大于 `tmp_table_size` 的时候，MyQL 也还是会在磁盘上创建临时表。

MySQL 不支持客户端的游标，不过客户端 API 可以通过缓存全部查询结果的方式模拟客户端的游标。这和直接将结果放在一个内存数组中来维护并没有什么不同。参考第 6 章，你可以看到更多关于一次性读取整个结果集到客户端时的性能。

291

7.6 绑定变量

从 MySQL 4.1 版本开始，就支持服务器端的绑定变量（`prepared statement`），这大大提高了客户端和服务器端数据传输的效率。你若使用一个支持新协议的客户端，如 MySQL C API，就可以使用绑定变量功能了。另外，Java 和 .NET 的也都可以使用各自的客户端 Connector/J 和 Connector/NET 来使用绑定变量。最后，还有一个 SQL 接口用于支持绑定变量，后面我们将讨论这个（这里容易引起困扰）。

当创建一个绑定变量 SQL 时，客户端向服务器发送了一个 SQL 语句的原型。服务器端收到这个 SQL 语句框架后，解析并存储这个 SQL 语句的部分执行计划，返回给客户端一个 SQL 语句处理句柄。以后每次执行这类查询，客户端都指定使用这个句柄。

绑定变量的 SQL，使用问号标记可以接收参数的位置，当真正需要执行具体查询的时候，则使用具体值代替这些问号。例如，下面是一个绑定变量的 SQL 语句：

```
INSERT INTO tbl(col1, col2, col3) VALUES (?, ?, ?);
```

可以通过向服务器端发送各个问号的取值和这个 SQL 的句柄来执行一个具体的查询。反复使用这样的方式执行具体的查询，这正是绑定变量的优势所在。具体如何发送取值参数和 SQL 句柄，则和各个客户端的编程语言有关。使用 Java 和 .NET 的 MySQL 连接器就是一种办法。很多使用 MySQL C 语言链接库的客户端可以提供类似的接口，需要根据使用的编程语言的文档来了解如何使用绑定变量。

因为如下的原因，MySQL 在使用绑定变量的时候可以更高效地执行大量的重复语句：

- 在服务器端只需要解析一次 SQL 语句。
- 在服务器端某些优化器的工作只需要执行一次，因为它会缓存一部分的执行计划。
- 以二进制的方式只发送参数和句柄，比起每次都发送 ASCII 码文本效率更高，一个二进制的日期字段只需要三个字节，但如果是 ASCII 码则需要十个字节。不过最大的节省还是来自于 BLOB 和 TEXT 字段，绑定变量的形式可以分块传输，而无须一次性传输。二进制协议在客户端也可能节省很多内存，减少了网络开销，另外，还节省了将数据从存储原始格式转换成文本格式的开销。
- 仅仅是参数——而不是整个查询语句——需要发送到服务器端，所以网络开销会更小。
- MySQL 在存储参数的时候，直接将其存放到缓存中，不再需要在内存中多次复制。 292

绑定变量相对也更安全。无须在应用程序中处理转义，一则更简单了，二则也大大减少了 SQL 注入和攻击的风险。(任何时候都不要信任用户输入，即使是使用绑定变量的时候。)

可以只在使用绑定变量的时候才使用二进制传输协议。如果使用普通的 mysql_query() 接口则不会使用二进制传输协议。还有一些客户端让你使用绑定变量，先发送带参数的绑定 SQL，然后发送变量值，但是实际上，这些客户端只是模拟了绑定变量的接口，最后还是会直接用具体值代替参数后，再使用 mysql_query() 发送整个查询语句。

7.6.1 绑定变量的优化

对使用绑定变量的 SQL，MySQL 能够缓存其部分执行计划，如果某些执行计划需要根据传入的参数来计算时，MySQL 就无法缓存这部分的执行计划。根据优化器什么时候工作，可以将优化分为三类。在本书编写的时候，下面的三点是适用的。

在准备阶段

服务器解析 SQL 语句，移除不可能的条件，并且重写子查询。

在第一次执行的时候

如果可能的话，服务器先简化嵌套循环的关联，并将外关联转化成内关联。

在每次 SQL 语句执行时

服务器做如下事情：

- 过滤分区。
- 如果可能的话，尽量移除 COUNT()、MIN() 和 MAX()。
- 移除常数表达式。
- 检测常量表。
- 做必要的等值传播。
- 分析和优化 ref、range 和索引优化等访问数据的方法。
- 优化关联顺序。

参考第 6 章，可以了解更多关于这些优化的信息。理论上，有些优化只需要做一次，但实际上，上面的操作还是都会被执行。

293

7.6.2 SQL 接口的绑定变量

在 4.1 和更新的版本中，MySQL 支持了 SQL 接口的绑定变量。不使用二进制传输协议也可以直接以SQL的方式使用绑定变量。下面案例展示了如何使用SQL接口的绑定变量：

```
mysql> SET @sql := 'SELECT actor_id, first_name, last_name
    -> FROM sakila.actor WHERE first_name = ?';
mysql> PREPARE stmt_fetch_actor FROM @sql;
mysql> SET @actor_name := 'Penelope';
mysql> EXECUTE stmt_fetch_actor USING @actor_name;
+----------+------------+-----------+
| actor_id | first_name | last_name |
+----------+------------+-----------+
|        1 | PENELOPE   | GUINESS   |
|       54 | PENELOPE   | PINKETT   |
|      104 | PENELOPE   | CRONYN    |
|      120 | PENELOPE   | MONROE    |
+----------+------------+-----------+
mysql> DEALLOCATE PREPARE stmt_fetch_actor;
```

当服务器收到这些 SQL 语句后，先会像一般客户端的链接库一样将其翻译成对应的操作。

这意味着你无须使用二进制协议也可以使用绑定变量。

正如你看到的，比起直接编写的 SQL 语句，这里的语法看起来有一些怪怪的。那么，这种写法实现的绑定变量到底有什么优势呢?

最主要的用途就是在存储过程中使用。在 MySQL 5.0 版本中，就可以在存储过程中使用绑定变量，其语法和前面介绍的 SQL 接口的绑定变量类似。这意味，可以在存储过程中构建并执行“动态”的 SQL 语句，这里的“动态”是指可以通过灵活地拼接字符串等参数构建 SQL 语句。例如，下面的示例存储过程中可以针对某个数据库执行 OPTIMIZE TABLE 的操作：

```
DROP PROCEDURE IF EXISTS optimize_tables;
DELIMITER //
CREATE PROCEDURE optimize_tables(db_name VARCHAR(64))
BEGIN
   DECLARE t VARCHAR(64);
   DECLARE done INT DEFAULT 0;
   DECLARE c CURSOR FOR
      SELECT table_name FROM INFORMATION_SCHEMA.TABLES
      WHERE TABLE_SCHEMA = db_name AND TABLE_TYPE = 'BASE TABLE';
   DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
   OPEN c;
   tables_loop: LOOP
      FETCH c INTO t;
      IF done THEN
         LEAVE tables_loop;
      END IF;
      SET @stmt_text := CONCAT("OPTIMIZE TABLE ", db_name, ".", t);
      PREPARE stmt FROM @stmt_text;
      EXECUTE stmt;
      DEALLOCATE PREPARE stmt;
   END LOOP;
   CLOSE c;
END//
DELIMITER ;
```

294

可以这样调用这个存储过程：

```
mysql> CALL optimize_tables('sakila');
```

另一种实现存储过程中循环的办法是：

```
REPEAT
   FETCH c INTO t;
   IF NOT done THEN
      SET @stmt_text := CONCAT("OPTIMIZE TABLE ", db_name, ".", t);
      PREPARE stmt FROM @stmt_text;
      EXECUTE stmt;
      DEALLOCATE PREPARE stmt;
   END IF;
UNTIL done END REPEAT;
```

这两种循环结构最重要的区别在于：REPEAT会为每个循环检查两次循环条件。在这个例子中，因为循环条件检查的是一个整数判断，并不会有什么性能问题，如果循环的判断条件非常复杂的话，则需要注意这两者的区别。

像这样使用 SQL 接口的绑定变量拼接表名和库名是很常见的，这样的好处是无须使用任何参数就能完成 SQL 语句。而库名和表名都是关键字，在二进制协议的绑定变量中是不能将这两部分参数化的。另一个经常需要动态设置的就是 LIMIT 子句，因为二进制协议中也无法将这个值参数化。

另外，编写存储过程时，SQL 接口的绑定变量通常可以很大程度地帮助我们调试绑定变量，如果不是在存储过程中，SQL 接口的绑定变量就不是那么有用了。因为 SQL 接口的绑定变量，它既没有使用二进制传输协议，也没有能够节省带宽，相反还总是需要增加至少一次额外网络传输才能完成一次查询。所有只有在某些特殊的场景下 SQL 接口的绑定变量才有用，比如当 SQL 语句非常非常长，并且需要多次执行的时候。

7.6.3 绑定变量的限制

关于绑定变量的一些限制和注意事项如下：

- 绑定变量是会话级别的，所以连接之间不能共用绑定变量句柄。同样地，一旦连接断开，则原来的句柄也不能再使用了。（连接池和持久化连接可以在一定程度上缓解这个问题。）
- 在 MySQL 5.1 版本之前，绑定变量的 SQL 是不能使用查询缓存的。
- 295 并不是所有的时候使用绑定变量都能获得更好的性能。如果只是执行一次 SQL，那么使用绑定变量方式无疑比直接执行多了一次额外的准备阶段消耗，而且还需要一次额外的网络开销。（要正确地使用绑定变量，还需要在使用完成后，释放相关的资源。）
- 当前版本下，还不能在存储函数中使用绑定变量（但是存储过程中可以使用）。
- 如果总是忘记释放绑定变量资源，则在服务器端很容易发生资源“泄漏”。绑定变量 SQL 总数的限制是一个全局限制，所以某一个地方的错误可能会对所有其他的线程都产生影响。
- 有些操作，如 BEGIN，无法在绑定变量中完成。

不过使用绑定变量最大的障碍可能是：它是如何实现以及原理是怎样的，这两点很容易让人困惑。有时，很难解释如下三种绑定变量类型之间的区别是什么：

客户端模拟的绑定变量

客户端的驱动程序接收一个带参数的 SQL，再将指定的值带入其中，最后将完整的

查询发送到服务器端。

服务器端的绑定变量

客户端使用特殊的二进制协议将带参数的字符串发送到服务器端，然后使用二进制协议将具体的参数值发送给服务器端并执行。

SQL 接口的绑定变量

客户端先发送一个带参数的字符串到服务器端，这类似于使用 PREPARE 的 SQL 语句，然后发送设置参数的 SQL，最后使用 EXECUTE 来执行 SQL。所有这些都使用普通的文本传输协议。

7.7 用户自定义函数

从很早开始，MySQL 就支持用户自定义函数（UDF）。存储过程只能使用 SQL 来编写，而 UDF 没有这个限制，你可以使用支持 C 语言调用约定的任何编程语言来实现。

UDF 必须事先编译好并动态链接到服务器上，这种平台相关性使得 UDF 在很多方面都很强大。UDF 速度非常快，而且可以访问大量操作系统的功能，还可以使用大量库函数。使用 SQL 实现的存储函数在实现一些简单操作上很有优势，诸如计算球体上两点之间的距离，但是如果操作涉及到网络交互，那么只能使用 UDF 了。同样地，如果需要一个 MySQL 不支持的统计聚合函数，而且无法使用 SQL 编写的存储函数来实现的话，通常使用 UDF 是很容易实现的。

能力越大，责任越大。所以在 UDF 中的一个错误很可能会让服务器直接崩溃，甚至扰乱服务器的内存或者数据，另外，所有 C 语言具有的潜在风险，UDF 也都有。

> 和使用 SQL 语言编写存储程序不同，UDF 无法读写数据表——至少，无法在调用 296
> UDF 的线程中使用当前事务处理的上下文来读写数据表。这意味着，它更适合用作计算或者与外面的世界交互。MySQL 已经支持越来越多的方式和外面的资源交互了。Brian Aker 和 Patrick Galbraith 创建的与 *memcached* 通信的函数就是一个 UDF 很好的案例（参考：*http://tangent.org/586/Memcached_Functions_for_MySQL.html*）。

如果打算使用 UDF，那么在 MySQL 版本升级的时候需要特别注意做相应的改变，因为很可能需要重新编译这些 UDF，或者甚至需要修改 UDF 来让它能在新的版本中工作。还需要注意的是，你需要确保 UDF 是线程安全的，因为它们需要在 MySQL 中执行，而 MySQL 是一个纯粹的多线程环境。

现在已经有很多写好的 UDF 直接提供给 MySQL 使用，还有很多 UDF 的示例可供参考，以便完成自己的 UDF。现在 UDF 最大的仓库是 *http://www.mysqludf.org*。

下面是一个用户自定义函数 NOW_USEC() 的代码，这个函数在第 10 章中我们将用它来测量复制的速度：

```
#include <my_global.h>
#include <my_sys.h>
#include <mysql.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>
extern "C" {
   my_bool now_usec_init(UDF_INIT *initid, UDF_ARGS *args, char *message);
   char *now_usec(
                  UDF_INIT *initid,
                  UDF_ARGS *args,
                  char *result,
                  unsigned long *length,
                  char *is_null,
                  char *error);
}
my_bool now_usec_init(UDF_INIT *initid, UDF_ARGS *args, char *message) {
   return 0;
}
char *now_usec(UDF_INIT *initid, UDF_ARGS *args, char *result,
               unsigned long *length, char *is_null, char *error) {
  struct timeval tv;
  struct tm* ptm;
  char time_string[20]; /* e.g. "2006-04-27 17:10:52" */
  char *usec_time_string = result;
  time_t t;
  /* Obtain the time of day, and convert it to a tm struct. */
  gettimeofday (&tv, NULL);
  t = (time_t)tv.tv_sec;
  ptm = localtime (&t);
  /* Format the date and time, down to a single second. */
  strftime (time_string, sizeof (time_string), "%Y-%m-%d %H:%M:%S", ptm);
  /* Print the formatted time, in seconds, followed by a decimal point
   * and the microseconds. */
  sprintf(usec_time_string, "%s.%06ld\n", time_string, tv.tv_usec);
  *length = 26;
  return(usec_time_string);
}
```

297

参考前一章中的案例学习，可以看到如何使用用户自定义函数来解决一些棘手的问题。我们在 Percona Toolkit 中也使用了 UDF 来完成一些工作，例如高效的数据复制校验，或者在 Sphinx 索引之前使用 UDF 来预处理一些问题等。UDF 是一款非常强大的工具。

7.8 插件

除了 UDF，MySQL 还支持各种各样的插件。这些插件可以在 MySQL 中新增启动选项和状态值，还可以新增 INFORMATION_SCHEMA 表，或者在 MySQL 的后台执行任务，等等。

在 MySQL 5.1 和更新的版本中，MySQL 新增了很多的插件接口，使得你无须直接修改 MySQL 的源代码就可以大大扩展它的功能。下面是一个简单的插件列表。

存储过程插件

存储过程插件可以帮你在存储过程运行后再处理一次运行结果。这是一个很古老的插件了，和 UDF 有些类似，多数人都可能忘记了这个插件的存在。内置的 PROCEDURE ANALYSE 就是一个很好的示例。

后台插件

后台插件可以让你的程序在 MySQL 中运行，可以实现自己的网络监听、执行自己的定期任务。后台插件的一个典型例子就是在 Percona Server 中包含的 Handler-Socket 插件。它监听一个新的网络端口，使用一个简单的协议可以帮你无须使用 SQL 接口直接访问 InnoDB 数据，这也使得 MySQL 能够像一些 NoSQL 一样具有非常高的性能。

INFORMATION_SCHEMA 插件

这个插件可以提供一个新的内存 INFORMATION_SCHEMA 表。

全文解析插件

这个插件提供一种处理文本的功能，可以根据自己的需求来对一个文档进行分词，所以如果给定一个 PDF 文档目录，可以使用这个插件对这个文档进行分词处理。也可以用此来增强查询执行过程中的词语匹配功能。

审计插件

审计插件在查询执行的过程中的某些固定点被调用，所以它可以用作（例如）记录 MySQL 的事件日志。

认证插件 298

认证插件既可以在 MySQL 客户端也可在它的服务器端，可以使用这类插件来扩展 MySQL 的认证功能，例如可以实现 PAM 和 LDAP 认证。

要了解更多细节，可以参考 MySQL 的官方手册，或者读读由 Sergei Golubchik 和 Andrew Hutchings (Packt) 编写的 *MySQL 5.1 Plugin Development*。如果你需要一个插件，但是却不知道怎么实现，有很多公司都提供这类咨询服务，例如 Monty Program、Open Query、Percona 和 SkySQL。

7.9 字符集和校对

字符集是指一种从二进制编码到某类字符符号的映射，可以参考如何使用一个字节来表示英文字母。“校对”是指一组用于某个字符集的排序规则。MySQL 4.1 和之后的版本中，

每一类编码字符都有其对应的字符集和校对规则[注9]。MySQL 对各种字符集的支持非常完善，但是这也带来了一定的复杂性，某些场景下甚至会有一定的性能牺牲。（另外，曾经 Drizzle 放弃了所有的字符集，所有字符全部统一使用 UTF-8。）

本节将解释在实际使用中，你可能最需要的一些设置和功能。如果想了解更多细节，可以详细地阅读 MySQL 官方手册的相关章节。

7.9.1 MySQL 如何使用字符集

每种字符集都可能有多种校对规则，并且都有一个默认的校对规则。每个校对规则都是针对某个特定的字符集的，和其他的字符集没有关系。校对规则和字符集总是一起使用的，所以后面我们将这样的组合也统称为一个字符集。

MySQL 有很多的选项用于控制字符集。这些选项和字符集很容易混淆，一定要记住：只有基于字符的值才真正的“有”字符集的概念。对于其他类型的值，字符集只是一个设置，指定用哪一种字符集来做比较或者其他操作。基于字符的值能存放在某列中、查询的字符串中、表达式的计算结果中或者某个用户变量中，等等。

MySQL 的设置可以分为两类：创建对象时的默认值、在服务器和客户端通信时的设置。

创建对象时的默认设置

MySQL 服务器有默认的字符集和校对规则，每个数据库也有自己的默认值，每个表也有自己的默认值。这是一个逐层继承的默认设置，最终最靠底层的默认设置将影响你创
299 建的对象。这些默认值，至上而下地告诉 MySQL 应该使用什么字符集来存储某个列。

在这个“阶梯”的每一层，你都可以指定一个特定的字符集或者让服务器使用它的默认值：

- 创建数据库的时候，将根据服务器上的 `character_set_server` 设置来设定该数据库的默认字符集。
- 创建表的时候，将根据数据库的字符集设置指定这个表的字符集设置。
- 创建列的时候，将根据表的设置指定列的字符集设置。

需要记住的是，真正存放数据的是列，所以更高“阶梯”的设置只是指定默认值。一个表的默认字符集设置无法影响存储在这个表中某个列的值。只有当创建列而没有为列指定字符集的时候，如果没有指定字符集，表的默认字符集才有作用。

注 9： MySQL 4.0 和更早的版本中，如果设置服务器的全局设置，有几种 8 字节的字符集可以选择。

服务器和客户端通信时的设置

当服务器和客户端通信的时候，它们可能使用不同的字符集。这时，服务器端将进行必要的翻译转换工作：

- 服务器端总是假设客户端是按照 character_set_client 设置的字符来传输数据和 SQL 语句的。
- 当服务器收到客户端的 SQL 语句时，它先将其转换成字符集 character_set_connection。它还使用这个设置来决定如何将数据转换成字符串。
- 当服务器端返回数据或者错误信息给客户端时，它会将其转换成 character_set_result。

图 7-2 展示了这个过程。

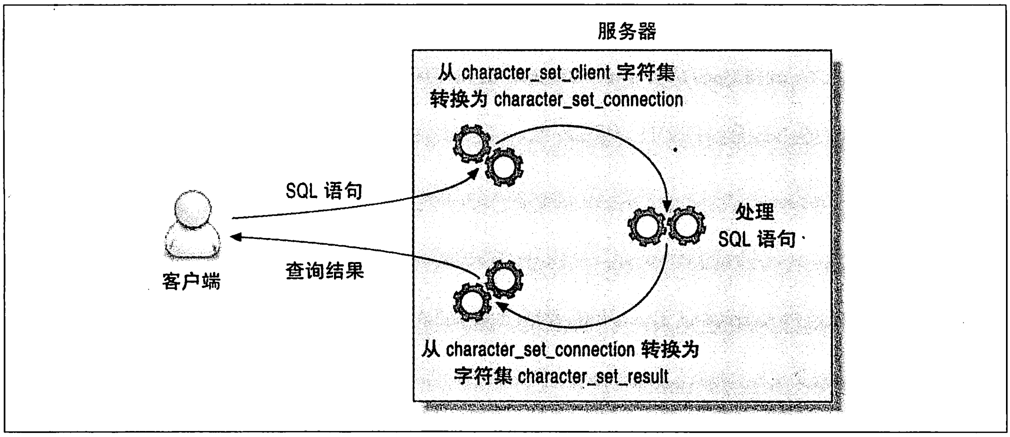


图7-2：客户端和服务器的字符集

根据需要，可以使用 SET NAMES 或者 SET CHARACTER SET 语句来改变上面的设置。不过在服务器上使用这个命令只能改变服务器端的设置。客户端程序和客户端的 API 也需要使用正确的字符集才能避免在通信时出现问题。 300

假设使用 latin1 字符集（这是默认字符集）打开一个连接，并使用 SET NAMES utf8 来告诉服务器客户端将使用 UTF-8 字符集来传输数据。这样就创建了一个不匹配的字符集，可能会导致一些错误甚至出现一些安全性问题。应当先设置客户端字符集然后使用函数 mysql_real_escape_string() 在需要的时候进行转义。在 PHP 中，可以使用 mysql_set_charset() 来修改客户端的字符集。

MySQL 如何比较两个字符串的大小

如果比较的两个字符串的字符集不同，MySQL 会先将其转成同一个字符集再进行比较。如果两个字符集不兼容的话，则会抛出错误，例如“ERROR 1267(HY000):Illegal mix of collations”。这种情况下需要通过函数 CONVERT() 显式地将其中一个字符串的字符集转成一个兼容的字符集。MySQL 5.0 和更新的版本经常会做这样的隐式转换，所以这类错误通常是在 MySQL 4.1 中比较常见。

MySQL 还会为每个字符串设置一个“可转换性”[注10]。这个设置决定了值的字符集的优先级，因而会影响 MySQL 做字符集隐式转换后的值。另外，也可以使用函数 CHARSET()、COLLATION()、和 COERCIBILITY() 来定位各种字符集相关的错误。

还可以使用前缀和 COLLATE 子句来指定字符串的字符集或者校对字符集。例如，下面的示例中使用了前缀（由下画线开始）来指定 utf8 字符集，还使用了 COLLATE 子句指定了使用二进制校对规则：

```
mysql> SELECT _utf8 'hello world' COLLATE utf8_bin;
+--------------------------------------+
| _utf8 'hello world' COLLATE utf8_bin |
+--------------------------------------+
| hello world                          |
+--------------------------------------+
```

一些特殊情况

MySQL 的字符集行为中还是有一些隐藏的“惊喜”的。下面列举了一些需要注意的地方：

诡异的 character_set_database 设置

character_set_database 设置的默认值和默认数据库的设置相同。当改变默认数据库的时候，这个变量也会跟着变。所以当连接到 MySQL 实例上又没有指定要使用的数据库时，默认值会和 character_set_server 相同。

301 LOAD DATA INFILE

当使用 LOAD DATA INFILE 的时候，数据库总是将文件中的字符按照字符集 character_set_database 来解析。在 MySQL 5.0 和更新的版本中，可以在 LOAD DATA INFILE 中使用子句 CHARACTER SET 来设定字符集，不过最好不要依赖这个设定。我们发现指定字符集最好的方式是先使用 USE 指定数据库，再执行 SET NAMES 来设定字符集，最后再加载数据。MySQL 在加载数据的时候，总是以同样的字符集处理所有数据，而不管表中的列是否有不同的字符集设定。

SELECT INTO OUTFILE

MySQL 会将 SELECT INTO OUTFILE 的结果不做任何转码地写入文件。目前，除了使

注 10： coercibility() 函数的返回值。——译者注

用函数 CONVERT() 将所有的列都做一次转码外，还没有什么别的办法能够指定输出的字符集。

嵌入式转义序列

MySQL 会根据 character_set_client 的设置来解析转义序列，即使是字符串中包含前缀或者 COLLATE 子句也一样。这是因为解析器在处理字符串中的转义字符时，完全不关心校对规则——对解析器来说，前缀并不是一个指令，它只是一个关键字而已。

7.9.2 选择字符集和校对规则

MySQL 4.1 和之后的版本支持很多的字符集和校对规则，包括支持使用 Unicode 编码的多字节 UTF-8 字符集（MySQL 支持 UTF-8 的一个三字节子集，这几乎可以包含世界上的所有字符集）。可以使用命令 SHOW CHARACTERSET 和 SHOW COLLATION 来查看 MySQL 支持的字符集和校对规则。

极简原则

在一个数据库中使用多个不同的字符集是一件很让人头疼的事情，字符集之间的不兼容问题会很难缠。有时候，一切都看起来正常，但是当某个特殊字符出现的时候，所有类型的操作都可能会无法进行（例如多表之间的关联）。你可以使用 ALTER TABLE 命令将对应列转成相互兼容的字符集，还可以使用编码前缀和 COLLATE 子句将对应的列值转成兼容的编码。

正确的方法是，最好先为服务器（或者数据库）选择一个合理的字符集。然后根据不同的实际情况，让某些列选择合适的字符集。

对于校对规则通常需要考虑的一个问题是，是否以大小写敏感的方式比较字符串，或者是以字符串编码的二进制值来比较大小。它们对应的校对规则的前缀分别是 _cs、_ci 和 _bin，根据需要很容易选择。大小写敏感和二进制校对规则的不同之处在于，二进制校对规则直接使用字符的字节进行比较，而大小写敏感的校对规则在多字节字符集时，如德语，有更复杂的比较规则。 302

在显式设置字符集的时候，并不是必须同时指定字符集和校对规则的名字。如果缺失了其中一个或者两个，MySQL 会使用可能的默认值来进行填充。表 7-2 表示了 MySQL 如何选择字符集和校对规则。

表7-2：MySQL如何选择字符集和校对规则

用户设置	返回结果的字符集	返回结果的校对规则
同时设置字符集和校对规则	与用户设置相同	与用户设置相同
仅设置字符集	与用户设置相同	与字符集的默认校对规则相同
仅设置校对规则	与校对规则对应的字符集相同	与用户设置相同
都未设置	使用默认值	使用默认值

下面的命令展示了在创建数据库、表、列的时候如何显式地指定字符集和校对规则：

```
CREATE DATABASE d CHARSET latin1;
CREATE TABLE d.t(
   col1 CHAR(1),
   col2 CHAR(1) CHARSET utf8,
   col3 CHAR(1) COLLATE latin1_bin
) DEFAULT CHARSET=cp1251;
```

这个表最后的字符集和校对规则如下：

```
mysql> SHOW FULL COLUMNS FROM d.t;
+------+---------+-------------------+
|Field | Type    | Collation         |
+------+---------+-------------------+
|col1  | char(1) | cp1251_general_ci |
|col2  | char(1) | utf8_general_ci   |
|col3  | char(1) | latin1_bin        |
+------+---------+-------------------+
```

7.9.3 字符集和校对规则如何影响查询

某些字符集和校对规则可能会需要更多的 CPU 操作，可能会消耗更多的内存和存储空间，甚至还会影响索引的正常使用。所以在选择字符集的时候，也有一些需要注意的地方。

不同的字符集和校对规则之间的转换可能会带来额外的系统开销。例如，数据表 sakila.film 在列 title 上有索引，可以加速下面的 ORDER BY 查询：

303

```
mysql> EXPLAIN SELECT title, release_year FROM sakila.film ORDER BY title\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: index
possible_keys: NULL
          key: idx_title
      key_len: 767
          ref: NULL
         rows: 953
        Extra:
```

只有排序查询要求的字符集与服务器数据的字符集相同的时候，才能使用索引进行排序。索引根据数据列的校对规则[注11]进行排序，这里使用的是 utf8_general_ci。如果希望使用别的校对规则进行排序，那么 MySQL 就需要使用文件排序：

```
mysql> EXPLAIN SELECT title, release_year
    -> FROM sakila.film ORDER BY title COLLATE utf8_bin\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 953
        Extra: Using filesort
```

为了能够适应各种字符集，包括客户端字符集、在查询中显式指定的字符集，MySQL 会在需要的时候进行字符集转换。例如，当使用两个字符集不同的列来关联两个表的时候，MySQL 会尝试转换其中一个列的字符集。这和在数据列外面封装一个函数一样，会让 MySQL 无法使用这个列上的索引。如果你不确定 MySQL 内部是否做了这种转换，可以在 EXPLAIN EXTENDED 后使用 SHOW WARNINGS 来查看 MySQL 是如何处理的。从输出中可以看到查询中使用的字符集，也可以看出 MySQL 是否做了字符集转换操作。

UTF-8 是一种多字节编码，它存储一个字符会使用变长的字节数（一到三个字节）。在 MySQL 内部，通常使用一个定长的空间来存储字符串，再进行相关操作，这样做的目的是希望总是保证缓存中有足够的空间来存储字符串。例如，一个编码是 UTF-8 的 CHAR(10) 需要 30 个字节，即使最终存储的时候没有存储任何“多字节”字符也是一样。变长的字段类型（VARCHAR TEXT）存储在磁盘上时不会有这个困扰，但当它存储在临时表中用来处理或者排序时，也总是会分配最大可能的长度。

在多字节字符集中，一个字符不再是一个字节。所以，在 MySQL 中有两个函数 LENGTH() 和 CHAR_LENGTH() 来计算字符串的长度，在多字节字符集中，这两个函数的返回结果会不同。如果使用的是多字节字符集，那么确保在统计字符集的时候使用 CHAR_LENGTH()。（例如需要做 SUBSTRING() 操作的时候）。其实，在应用程序中也同样要注意多字节字符集的这个问题。 304

另一个“惊喜”可能是关于索引限制方面的。如果要索引一个 UTF-8 字符集的列，MySQL 会假设每一个字符都是三个字节，所以最长索引前缀的限制一下缩短到原来的三分之一了：

注 11： 即排序规则。——译者注

```
mysql> CREATE TABLE big_string(str VARCHAR(500), KEY(str)) DEFAULT CHARSET=utf8;
Query OK, 0 rows affected, 1 warning (0.06 sec)
mysql> SHOW WARNINGS;
+---------+------+---------------------------------------------------------+
| Level   | Code | Message                                                 |
+---------+------+---------------------------------------------------------+
| Warning | 1071 | Specified key was too long; max key length is 999 bytes |
+---------+------+---------------------------------------------------------+
```

注意到，MySQL 的索引前缀自动缩短到 333 个字符了：

```
mysql> SHOW CREATE TABLE big_string\G
*************************** 1. row ***************************
       Table: big_string
Create Table: CREATE TABLE `big_string` (
  `str` varchar(500) default NULL,
  KEY `str` (`str`(333))
) ENGINE=MyISAM DEFAULT CHARSET=utf8
```

如果你不注意警告信息也没有再重新检查表的定义，可能不会注意到这里仅仅是在该列的前缀上建立了索引。这会对 MySQL 使用索引有一些影响，例如无法使用索引覆盖扫描。

也有人建议，直接使用 UTF-8 字符集，"整个世界都清净了"。不过从性能的角度来看这不是一个好主意。根据存储的数据，很多应用无须使用 UTF-8 字符集，如果坚持使用 UTF-8，只会消耗更多的磁盘空间。

在考虑使用什么字符集的时候，需要根据存储的具体内存来决定。例如，存储的内容主要是英文字符，那么即使使用 UTF-8 也不会消耗太多的存储空间，因为英文字符在 UTF-8 字符集中仍然使用一个字节。但如果需要存储一些非拉丁语系的字符，如俄语、阿拉伯语，那么区别会很大。如果应用中只需要存储阿拉伯语，那么可以使用 cp1256 字符集，这个字符集可以用一个字节表示所有的阿拉伯语字符。如果还需要存储别的语言，那么就应该使用 UTF-8 了，这时相同的阿拉伯语字符会消耗更多的空间。类似地，当从某个具体的语种编码转换成 UTF-8 时，存储空间的使用会相应增加。如果使用的是 InnoDB 表，那么字符集的改变可能导致数据的大小超过可以在页内存储的临界值，需要保存在额外的外部存储区，这会导致很严重的空间浪费，还会带来很多空间碎片。

305 有时候根本不需要使用任何的字符集。通常只有在做大小写无关的比较、排序、字符串操作（例如 SUBSTRING() 的时候才需要使用字符集。如果你的数据库不关心字符集，那么可以直接将所有的东西存储到二进制列中，包括 UTF-8 编码数据也可以存储在其中。这么做，可能还需要一个列记录字符的编码集。虽然很多人一直都是这么用的，但还是有不少事项需要注意。这会导致很多难以排查的错误，例如，忘记了多个字节才是一个

字符时，还继续使用 SUBSTRING() 和 LENGTH() 做字符串操作，就会出错。如果可能，我们建议尽量不要这样做。

7.10 全文索引

通过数值比较、范围过滤等就可以完成绝大多数我们需要的查询了。但是，如果你希望通过关键字的匹配来进行查询过滤，那么就需要基于相似度的查询，而不是原来的精确数值比较。全文索引就是为这种场景设计的。

全文索引有着自己独特的语法。没有索引也可以工作，如果有索引效率会更高。用于全文搜索的索引有着独特的结构，帮助这类查询找到匹配某些关键字的记录。

你可能没有在意过全文索引，不过至少应该对一种全文索引技术比较熟悉：互联网搜索引擎。虽然这类搜索引擎的索引对象是超大量的数据，并且通常其背后都不是关系型数据库，不过全文索引的基本原理都是一样的。

全文索引可以支持各种字符内容的搜索（包括 CHAR、VARCHAR 和 TEXT 类型），也支持自然语言搜索和布尔搜索。在 MySQL 中全文索引有很多的限制[注 12]，其实现也很复杂，但是因为它是 MySQL 内置的功能，而且满足很多基本的搜索需求，所以它的应用仍然非常广泛。本章我们将介绍如何使用全文索引，以及如何为应用设计更高性能的全文索引。

在本书编写时，在标准的 MySQL 中，只有 MyISAM 引擎支持全文索引。不过在还没有正式发布的 MySQL 5.6 中，InnoDB 已经实验性质地支持全文索引了。除此，还有第三方的存储引擎，如 Groonga，也支持全文索引。

事实上，MyISAM 对全文索引的支持有很多的限制，例如表级别锁对性能的影响、数据
文件的崩溃、崩溃后的恢复等，这使得 MyISAM 的全文索引对于很多应用场景并不合适。
所以，多数情况下我们建议使用别的解决方案，例如 Sphinx、Lucene、Solr、Groonga、
Xapian 或者 Senna，再或者可以等 MySQL 5.6 版本正式发布后，直接使用 InnoDB 的全 306
文索引。如果 MyISAM 的全文索引确实能满足应用的需求，那么可以继续阅读本节。

MyISAM 的全文索引作用对象是一个“全文集合”，这可能是某个数据表的一列，也可能是多个列。具体的，对数据表的某一条记录，MySQL 会将需要索引的列全部拼接成一个字符串，然后进行索引。

注 12： 在 MySQL 5.1 中，可以使用全文解析器插件来扩展全文索引的功能。不过，MySQL 的全文索引本身还是有很多限制的，可能导致无法在你的应用场景中使用。我们将在附录 F 中介绍如何将 Sphinx 作为一个 MySQL 内部搜索引擎来使用。

MyISAM 的全文索引是一类特殊的 B-Tree 索引，共有两层。第一层是所有关键字，然后对于每一个关键字的第二层，包含的是一组相关的“文档指针”。全文索引不会索引文档对象中的所有词语，它会根据如下规则过滤一些词语：

- 停用词列表中的词都不会被索引。默认的停用词根据通用英语的使用来设置，可以使用参数 ft_stopword_file 指定一组外部文件来使用自定义的停用词。
- 对于长度大于 ft_min_word_len 的词语和长度小于 ft_max_word_len 的词语，都不会被索引。

全文索引并不会存储关键字具体匹配在哪一列，如果需要根据不同的列来进行组合查询，那么不需要针对每一列来建立多个这类索引。

这也意味着不能在 MATCH AGAINST 子句中指定哪个列的相关性更重要。通常构建一个网站的搜索引擎是需要这样的功能，例如，你可能希望优先搜索出那些在标题中出现过的文档对象。如果需要这样的功能，则需要编写更复杂的查询语句。（后面将会为大家展示如何实现。）

7.10.1 自然语言的全文索引

自然语言搜索引擎将计算每一个文档对象和查询的相关度。这里，相关度是基于匹配的关键词个数，以及关键词在文档中出现的次数。在整个索引中出现次数越少的词语，匹配时的相关度就越高。相反，非常常见的单词将不会搜索，即使不在停用词列表中出现，如果一个词语在超过 50% 的记录中都出现了，那么自然语言搜索将不会搜索这类词语。[注 13]

全文索引的语法和普通查询略有不同。可以根据 WHERE 子句中的 MATCH AGAINST 来区分查询是否使用全文索引。我们来看一个示例。在标准的数据库 Sakila 中，数据表 film_text 在字段 title 和 description 上建立了全文索引：

307
```
mysql> SHOW INDEX FROM sakila.film_text;
+-----------+-----------------------+-------------+------------+
| Table     | Key_name              | Column_name | Index_type |
+-----------+-----------------------+-------------+------------+
| ...
| film_text | idx_title_description | title       | FULLTEXT   |
| film_text | idx_title_description | description | FULLTEXT   |
+-----------+-----------------------+-------------+------------+
```

注 13：在测试使用时的一个常见错误就是，只是用很小的数据集合进行全文索引，所以总是无法返回结果。原因在于，每个搜索关键词都可能在一半以上的记录里面出现过。

下面是一个使用自然语言搜索的查询：

```
mysql> SELECT film_id, title, RIGHT(description, 25),
    ->    MATCH(title, description) AGAINST('factory casualties') AS relevance
    -> FROM sakila.film_text
    -> WHERE MATCH(title, description) AGAINST('factory casualties');
+---------+-----------------------+-------------------------+-----------------+
| film_id | title                 | RIGHT(description, 25)  | relevance       |
+---------+-----------------------+-------------------------+-----------------+
|     831 | SPIRITED CASUALTIES   | a Car in A Baloon Factory | 8.4692449569702 |
|     126 | CASUALTIES ENCINO     | Face a Boy in A Monastery | 5.2615661621094 |
|     193 | CROSSROADS CASUALTIES | a Composer in The Outback | 5.2072987556458 |
|     369 | GOODFELLAS SALUTE     | d Cow in A Baloon Factory | 3.1522686481476 |
|     451 | IGBY MAKER            | a Dog in A Baloon Factory | 3.1522686481476 |
```

MySQL 将搜索词语分成两个独立的关键词进行搜索，搜索在 title 和 description 字段组成的全文索引上进行。注意，只有一条记录同时包含全部的两个关键词，有三个查询结果只包含关键字"casualties"（这是整个表中仅有的三条包含该关键词的记录），这三个结果都在结果列表的前面。这是因为查询结果是根据与关键词的相似度来进行排序的。

> 和普通查询不同，这类查询自动按照相似度进行排序。在使用全文索引进行排序的时候，MySQL 无法再使用索引排序。所以如果不想使用文件排序的话，那么就不要在查询中使用 ORDER BY 子句。

从上面的示例可以看到，函数 MATCH() 将返回关键词匹配的相关度，是一个浮点数字。你可以根据相关度进行匹配，或者将此直接展现给用户。在一个查询中使用两次 MATCH() 函数并不会有额外的消耗，MySQL 会自动识别并只进行一次搜索。不过，如果你将 MATCH() 函数放到 ORDER BY 子句中，MySQL 将会使用文件排序。

在 MATCH() 函数中指定的列必须和在全文索引中指定的列完全相同，否则就无法使用全文索引。这是因为全文索引不会记录关键字是来自哪一列的。

这也意味着无法使用全文索引来查询某个关键字是否在某一列中存在。这里介绍一个绕过该问题的办法：根据关键词在多个不同列的全文索引上的相关度来算出排名值，然后 308
依此来排序。我们可以在某一列上加上如下索引：

```
mysql> ALTER TABLE film_text ADD FULLTEXT KEY(title) ;
```

这样，我们可以将 title 匹配乘以 2 来提高它的相似度的权重：

```
mysql> SELECT film_id, RIGHT(description, 25),
    -> ROUND(MATCH(title, description) AGAINST('factory casualties'), 3)
    ->    AS full_rel,
    -> ROUND(MATCH(title) AGAINST('factory casualties'), 3) AS title_rel
    -> FROM sakila.film_text
    -> WHERE MATCH(title, description) AGAINST('factory casualties')
    -> ORDER BY (2 * MATCH(title) AGAINST('factory casualties'))
    ->    + MATCH(title, description) AGAINST('factory casualties') DESC;
+---------+---------------------------+----------+-----------+
| film_id | RIGHT(description, 25)    | full_rel | title_rel |
+---------+-------------- ------------+----------+-----------+
|     831 | a Car in A Baloon Factory |    8.469 |     5.676 |
|     126 | Face a Boy in A Monastery |    5.262 |     5.676 |
|     299 | jack in The Sahara Desert |    3.056 |     6.751 |
|     193 | a Composer in The Outback |    5.207 |     5.676 |
|     369 | d Cow in A Baloon Factory |    3.152 |     0.000 |
|     451 | a Dog in A Baloon Factory |    3.152 |     0.000 |
|     595 | a Cat in A Baloon Factory |    3.152 |     0.000 |
|     649 | nizer in A Baloon Factory |    3.152 |     0.000 |
```

因为上面的查询需要做文件排序，所以这并不是一个高效的做法。

7.10.2 布尔全文索引

在布尔搜索中，用户可以在查询中自定义某个被搜索的词语的相关性。布尔搜索通过停用词列表过滤掉那些“噪声”词，除此之外，布尔搜索还要求搜索关键词长度必须大于 ft_min_word_len，同时小于 ft_max_word_len[注14]。搜索返回的结果是未经排序的。

当编写一个布尔搜索查询时，可以通过一些前缀修饰符来定制搜索。表 7-3 列出了最常用的修饰符。

表7-3：布尔全文索引通用修饰符

Example	Meaning
dinosaur	包含“dinosaur”的行 rank 值更高
~dinosaur	包含“dinosaur”的行 rank 值更低
+dinosaur	行记录必须包含“dinosaur”
309 -dinosaur	行记录不可以包含“dinosaur”
dino*	包含以“dino”开头的单词的行 rank 值更高

还可以使用其他的操作，例如使用括号分组。基于此，就可以构造出一些复杂的搜索查询。

还是继续用 sakila.film_text 来举例，现在我们需要搜索既包含词“factory”又包含“casualties”的记录。在前面，我们已经使用自然语言搜索查询实现找到这两个词中的

注 14： 事实上，全文索引根本不会对太短或者太长的词语进行索引，但是这里说的不是一回事。一般地，MySQL 本身并不会因为搜索关键词过长或过短而忽略这些词语，但是查询优化器的某些部分却可能这样做。

任何一个的 SQL 写法。使用布尔搜索查询，我们可以指定返回结果必须同时包含“factory”和“casualties”：

```
mysql> SELECT film_id, title, RIGHT(description, 25)
    -> FROM sakila.film_text
    -> WHERE MATCH(title, description)
    ->    AGAINST('+factory +casualties' IN BOOLEAN MODE);
+---------+--------------------+--------------------------+
| film_id | title              | RIGHT(description, 25)   |
+---------+--------------------+--------------------------+
|     831 | SPIRITED CASUALTIES | a Car in A Baloon Factory |
+---------+--------------------+--------------------------+
```

查询中还可以使用括号进行“短语搜索”，让返回结果精确匹配指定的短语：

```
mysql> SELECT film_id, title, RIGHT(description, 25)
    -> FROM sakila.film_text
    -> WHERE MATCH(title, description)
    ->    AGAINST('"spirited casualties"' IN BOOLEAN MODE);
+---------+--------------------+--------------------------+
| film_id | title              | RIGHT(description, 25)   |
+---------+--------------------+--------------------------+
|     831 | SPIRITED CASUALTIES | a Car in A Baloon Factory |
+---------+--------------------+--------------------------+
```

短语搜索的速度会比较慢。只使用全文索引是无法判断是否精确匹配短语的，通常还需要查询原文确定记录中是否包含完整的短语。由于需要进行回表过滤，所以速度会很慢。

要完成上面的查询，MySQL 需要先从索引中找出所有同时包含“spirited”和“casualties”的索引条目，然后取出这些记录再判断是否是精确匹配短语。因为这个操作会先从索引中过滤出一些记录，所以通常认为这样做的速度是很快的——比 LIKE 操作要快很多。事实上，这样做的确很快，但是搜索的关键词不能是太常见的词语。如果搜索的关键词太常见，因为前一步的过滤会返回太多的记录需要判断，因此 LIKE 操作反而更快。这种情况下 LIKE 操作是完全的顺序读，相比索引返回值的随机读，会快很多。

只有 MyISAM 引擎才能使用布尔全文索引，但并不是一定要有全文索引才能使用布尔全文搜索。当没有全文索引的时候，MySQL 就通过全表扫描来实现。所以，你甚至还可以在多表上使用布尔全文索引，例如在一个关联结果上进行。只不过，因为是全表扫描，速度可能会很慢。 310

7.10.3 MySQL 5.1 中全文索引的变化

在 MySQL 5.1 中引入了一些和全文索引相关的改进，包括一些性能上的提升和新增插件式的解析，通过此用户可以自己定制增强搜索功能。例如，插件可以改变索引文本的方式。可以用更灵活的方式进行分词（例如，可以指定 C++ 作为一个单独的词语）、预处理、

可以对不同的文档类型进行索引（如 PDF），还可以做一些自定义的词干规则。插件还可以直接影响全文搜索的工作方式——例如，直接使用词干进行搜索。

7.10.4 全文索引的限制和替代方案

MySQL 的全文索引实现有很多的设计本身带来的限制。在某些场景下这些限制是致命的，不过也有很多办法绕过这些限制。

例如，MySQL 全文索引中只有一种判断相关性的方法：词频。索引也不会记录索引词在字符串中的位置，所以位置也就无法用在相关性上。虽然大多数情况下，尤其是数据量很小的时候，这些限制都不会影响使用，但也可能不是你所想要的。而且 MySQL 的全文索引也没有提供其他可选的相关性排序算法。（它无法存储基于相对位置的相关性排序数据。）

数据量的大小也是一个问题。MySQL 的全文索引只有全部在内存中的时候，性能才非常好。如果内存无法装载全部索引，那么搜索速度可能会非常慢。当你使用精确短语搜索时，想要好的性能，数据和索引都需要在内存中。相比其他的索引类型，当 INSERT、UPDATE 和 DELETE 操作进行时，全文索引的操作代价都很大：

- 修改一段文本中的 100 个单词，需要 100 次索引操作，而不是一次。
- 一般来说列长度并不会太影响其他的索引类型，但是如果是全文索引，三个单词的文本和 10 000 个单词的文本，性能可能会相差几个数量级。
- 全文索引会有更多的碎片，可能需要做更多的 OPTIMIZE TABLE 操作。

全文索引还会影响查询优化器的工作。索引选择、WHERE 子句、ORDER BY 都有可能不是按照你所预想的方式来工作：

311
- 如果查询中使用了 MATCH AGAINST 子句，而对应列上又有可用的全文索引，那么 MySQL 就一定会使用这个全文索引。这时，即使有其他的索引可以使用，MySQL 也不会去比较到底哪个索引的性能更好。所以，即使这时有更合适的索引可以使用，MySQL 仍然会置之不理。
- 全文索引只能用作全文搜索匹配。任何其他操作，如 WHERE 条件比较，都必须在 MySQL 完成全文搜索返回记录后才能进行。这和其他普通索引不同，例如，在处理 WHERE 条件时，MySQL 可以使用普通索引一次判断多个比较表达式。
- 全文索引不存储索引列的实际值。也就不可能用作索引覆盖扫描。
- 除了相关性排序，全文索引不能用作其他的排序。如果查询需要做相关性以外的排序操作，都需要使用文件排序。

让我们看看这些限制如何影响查询语句。来看一个例子，假设有一百万个文档记录，在文档的作者 author 字段上有一个普通的索引，在文档内容字段 content 上有全文索引。现在我们要搜索作者是 123，文档中又包含特定词语的文档。很多人可能会按照下面的方式来写查询语句：

```
... WHERE MATCH(content) AGAINST ('High Performance MySQL')
    AND author = 123;
```

而实际上，这样做的效率非常低。因为这里使用了 MATCH AGAINST，而且恰好上面有全文索引，所以 MySQL 优先选择使用全文索引，即先搜索所有的文档，查找是否有包含关键词的文档，然后返回记录看看作者是否是 123。所以这里也就没有使用 author 字段上的索引。

一个替代方案是将 author 列包含到全文索引中。可以在 author 列的值前面附上一个不常见的前缀，然后将这个带前缀的值存放到一个单独的 filters 列中，并单独维护该列（也许可以使用触发器来做维护工作）。

这样就可以扩展全文索引，使其包含 filters 列，上面的查询就可以改写为：

```
... WHERE MATCH(content, filters)
    AGAINST ('High Performance MySQL +author_id_123' IN BOOLEAN MODE);
```

这个案例中，如果 author 列的选择性非常高，那么 MySQL 能够根据作者信息很快地将需要过滤的文档记录限制在一个很小的范围内，这个查询的效率也就会非常好。如果 author 列的选择性很低，那么这个替代方案的效率会比前面那个更糟，所以使用的时候要谨慎。

全文索引有时候还可以实现一些简单的“边框”搜索。例如，希望搜索某个坐标范围时，将坐标按某种方式转换成文本再进行全文索引。假设某条记录的坐标为 X=123 和 Y=456。可以按照这样的方式交错存储坐标：XY142536，然后对此进行全文索引。这时， 312
希望查询某矩形——X 取值 100 至 199，Y 取值 400 至 499——范围时，可以在查询直接搜索“+XY14*”。这比使用 WHERE 条件过滤的效率要高很多。

全文索引的另一个常用技巧是缓存全文索引返回的主键值，这在分页显示的时候经常使用。当应用程序真的需要输出结果时，才通过主键值将所有需要的数据返回。这个查询就可以自由地使用其他索引、或者自由地关联其他表。

虽然只有 MyISAM 表支持全文索引，但是如果仍然希望使用 InnoDB 或其他引擎，可以将原表复制到一个备库，再将备库上的表改成 MyISAM 并建上相应的全文索引。如果不希望在另一个服务器上完成查询，还可以对表进行垂直拆分，将需要索引的列放到一个

单独的 MyISAM 表中。

将需要索引的列额外地冗余在另一个 MyISAM 表中也是一个办法。在测试库中 sakila.film_text 就是使用这个策略，这里使用触发器来维护这个表的数据。最后，你还可以使用一个包含内置全文索引的引擎，如 Lucene 或者 Sphinx。更多关于 Shpinx 的内容请参考附录 F。

因为使用全文索引的时候，通常会返回大量结果并产生大量随机 I/O，如果和 GROUP BY 一起使用的话，还需要通过临时表或者文件排序进行分组，性能会非常非常糟糕。这类查询通常只是希望查询分组后的前几名结果，所以一个有效的优化方法是对结果集进行抽样而不是精确计算。例如，仅查询前面的 1 000 条记录，进行分组并返回前几名的结果。

7.10.5 全文索引的配置和优化

全文索引的日常维护通常能够大大提升性能。“双 B-Tree”的特殊结构、在某些文档中比其他文档要包含多得多的关键字，这都使得全文索引比起普通索引有更多的碎片问题。所以需要经常使用 OPTIMIZE TABLE 来减少碎片。如果应用是 I/O 密集型的，那么定期地进行全文索引重建可以让性能提升很多。

如果希望全文索引能够高效地工作，还需要保证索引缓存足够大，从而保证所有的全文索引都能够缓存在内存中。通常，可以为全文索引设置单独的键缓存（Key cache），保证不会被其他的索引缓存挤出内存。键缓存的配置和使用可以参考第 8 章。

提供一个好的停用词表也很重要。默认的停用词表对常用英语来说可能还不错，但是如果是其他语言或者某些专业文档就不合适了，例如技术文档。例如，若要索引一批
313 MySQL 相关的文档，那么最好将 mysql 放入停用词表，因为在这类文档中，这个词会出现得非常频繁。

忽略一些太短的单词也可以提升全文索引的效率。索引单词的最小长度可以通过参数 ft_min_word_len 配置。修改该参数可以过滤更多的单词，让查询速度更快，但是也会降低精确度。还需要注意一些特殊的场景，有时确实需要索引某些非常短的词语。例如，对一个电子消费品文档进行索引，除非我们允许对很短的单词进行索引，否则搜索“cd player”可能会返回大量的结果。因为单词“cd”比默认允许的最短长度 4 还要小，所以这里只会对“Player”进行搜索，而通常搜索“cd player”的客户，其实对 MP3 或者 DVD 播放器并不感兴趣。

停用词表和允许最小词长都可以通过减少索引词语来提升全文索引的效率，但是同时也会降低搜索的精确度。这需要根据实际的应用场景找到合适的平衡点。如果你希望同时获得好的性能和好的搜索质量，那么需要自己定制这些参数。一个好的办法是通过日志

系统来研究用户的搜索行为，看看一些异常的查询，包括没有结果返回的查询或者返回过多结果的用户查询。通过这些用户行为和被搜索的内容来判断应该如何调整索引策略。

> 需要注意，当调整“允许最小词长”后，需要通过 OPTIMIZE TABLE 来重建索引才会生效。另一个参数 ft_max_word_len 和该参数行为类似，它限制了允许索引的最大词长。

当向一个有全文索引的表中导入大量数据的时候，最好先通过命令 DISABLE KEYS 来禁用全文索引，然后在导入结束后使用 ENABLE KYES 来建立全文索引。因为全文索引的更新是一个消耗很大的操作，所以上面的细节会帮你节省大量时间。另外，这样还顺便为全文索引做了一次碎片整理工作。

如果数据集特别大，则需要对数据进行手动分区，然后将数据分布到不同的节点，再做并行的搜索。这是一个复杂的工作，最好通过一些外部的搜索引擎来实现，如 Lucene 或者 Sphinx。我们的经验显示这样做性能会有指数级的提升。

7.11 分布式（XA）事务

存储引擎的事务特性能够保证在存储引擎级别实现 ACID（参考前面介绍的“事务”），而分布式事务则让存储引擎级别的 ACID 可以扩展到数据库层面，甚至可以扩展到多个数据库之间——这需要通过两阶段提交实现。MySQL 5.0 和更新版本的数据库已经开始 314
支持 XA 事务了。

XA 事务中需要有一个事务协调器来保证所有的事务参与者都完成了准备工作（第一阶段）。如果协调器收到所有的参与者都准备好的消息，就会告诉所有的事务可以提交了，这是第二阶段。MySQL 在这个 XA 事务过程中扮演一个参与者的角色，而不是协调者。

实际上，在 MySQL 中有两种 XA 事务。一方面，MySQL 可以参与到外部的分布式事务中；另一方面，还可以通过 XA 事务来协调存储引擎和二进制日志。

7.11.1 内部 XA 事务

MySQL 本身的插件式架构导致在其内部需要使用 XA 事务。MySQL 中各个存储引擎是完全独立的，彼此不知道对方的存在，所以一个跨存储引擎的事务就需要一个外部的协调者。如果不使用 XA 协议，例如，跨存储引擎的事务提交就只是顺序地要求每个存储引擎各自提交。如果在某个存储提交过程中发生系统崩溃，就会破坏事务的特性（要么就全部提交，要么就不做任何操作）。

如果将 MySQL 记录的二进制日志操作看作一个独立的“存储引擎”，就不难理解为什么即使是一个存储引擎参与的事务仍然需要 XA 事务了。在存储引擎提交的同时，需要将“提交”的信息写入二进制日志，这就是一个分布式事务，只不过二进制日志的参与者是 MySQL 本身。

XA 事务为 MySQL 带来巨大的性能下降。从 MySQL 5.0 开始，它破坏了 MySQL 内部的“批量提交”（一种通过单磁盘 I/O 操作完成多个事务提交的技术），使得 MySQL 不得不进行多次额外的 fsync() 调用[注 15]。具体的，一个事务如果开启了二进制日志，则不仅需要对二进制日志进行持久化操作，InnoDB 事务日志还需要两次日志持久化操作。换句话说，如果希望有二进制日志安全的事务实现，则至少需要做三次 fsync() 操作。唯一避免这个问题的办法就是关闭二进制日志，并将 innodb_support_xa 设置为 0[注 16]。

315 但这样的设置是非常不安全的，而且这会导致 MySQL 复制也没法正常工作。复制需要二进制日志和 XA 事务的支持，另外——如果希望数据尽可能安全——最好还要将 sync_binlog 设置成 1，这时存储引擎和二进制日志才是真正同步的。（否则，XA 事务支持就没有意义了，因为事务提交了二进制日志却可能没有“提交”到磁盘。）这也是为什么我们强烈建议使用带电池保护的 RAID 卡写缓存 ： 这个缓存可以大大加快 fsync() 操作的效率。

下一章我们将更进一步地介绍如何配置事务日志和二进制日志。

7.11.2 外部 XA 事务

MySQL 能够作为参与者完成一个外部的分布式事务。但它对 XA 协议支持并不完整，例如，XA 协议要求在一个事务中的多个连接可以做关联，但目前的 MySQL 版本还不能支持。

因为通信延迟和参与者本身可能失败，所以外部 XA 事务比内部消耗会更大。如果在广域网中使用 XA 事务，通常会因为不可预测的网络性能导致事务失败。如果有太多不可控因素，例如，不稳定的网络通信或者用户长时间地等待而不提交，则最好避免使用 XA 事务。任何可能让事务提交发生延迟的操作代价都很大，因为它影响的不仅仅是自己本身，它还会让所有参与者都在等待。

通常，还可以使用别的方式实现高性能的分布式事务。例如，可以在本地写入数据，并

注 15： 在撰写本书的时候，“批量提交”的问题已经有了很多解决方案，其中至少有三种是很优秀的。还需要进一步观察到底 MySQL 官方会采用哪一种，到底到哪个版本 MySQL 才会合并到源码。目前，使用 MariaDB 和 Percona Server 就可以避免这个问题。

注 16： 一个常见的误区是认为 innodb_support_xa 只有在需要 XA 事务时才需要打开。这是错误的 ：该参数还会控制 MyQSL 内部存储引擎和二进制日志之间的分布式事务。如果你真正关心你的数据，你需要将这个参数打开。

将其放入队列，然后在一个更小、更快的事务中自动分发。还可以使用 MySQL 本身的复制机制来发送数据。我们看到很多应用程序都可以完全避免使用分布式事务。

也就是说，XA 事务是一种在多个服务器之间同步数据的方法。如果由于某些原因不能使用 MySQL 本身的复制，或者性能并不是瓶颈的时候，可以尝试使用。

7.12 查询缓存

很多数据库产品都能够缓存查询的执行计划，对于相同类型的 SQL 就可以跳过 SQL 解析和执行计划生成阶段。MySQL 在某些场景下也可以实现，但是 MySQL 还有另一种不同的缓存类型：缓存完整的 SELECT 查询结果，也就是“查询缓存”。本节将详细介绍这类缓存。

MySQL查询缓存保存查询返回的完整结果。当查询命中该缓存，MySQL会立刻返回结果，跳过了解析、优化和执行阶段。

查询缓存系统会跟踪查询中涉及的每个表，如果这些表发生变化，那么和这个表相关的所有的缓存数据都将失效。这种机制效率看起来比较低，因为数据表变化时很有可能对应的查询结果并没有变更，但是这种简单实现代价很小，而这点对于一个非常繁忙的系统来说非常重要。 316

查询缓存对应用程序是完全透明的。应用程序无须关心 MySQL 是通过查询缓存返回的结果还是实际执行返回的结果。事实上，这两种方式执行的结果是完全相同的。换句话说，查询缓存无须使用任何语法。无论是 MySQL 开启或关闭查询缓存，对应用程序都是透明的[注17]。

随着现在的通用服务器越来越强大，查询缓存被发现是一个影响服务器扩展性的因素。它可能成为整个服务器的资源竞争单点，在多核服务器上还可能导致服务器僵死。后面我们将详细介绍如何配合查询缓存，但是很多时候我们还是认为应该默认关闭查询缓存，如果查询缓存作用很大的话，那就配置一个很小的查询缓存空间（如几十兆）。后面我们将解释如何判断在你的系统压力下打开查询缓存是否有好处。

7.12.1 MySQL 如何判断缓存命中

MySQL 判断缓存命中的方法很简单：缓存存放在一个引用表中，通过一个哈希值引用，这个哈希值包括了如下因素，即查询本身、当前要查询的数据库、客户端协议的版本等

注 17：有一种方式查询缓存可能和原生的 SQL 工作方式有所不同：默认的，当要查询的表被 LOCK TABLES 锁住时，查询仍然可以通过查询缓存返回数据。你可以通过参数 query_cache_wlock_invalidate 打开或者关闭这种行为。

一些其他可能会影响返回结果的信息。

当判断缓存是否命中时，MySQL 不会解析、"正规化"或者参数化查询语句，而是直接使用 SQL 语句和客户端发送过来的其他原始信息。任何字符上的不同，例如空格、注释——任何的不同——都会导致缓存的不命中。[注18] 所以在编写 SQL 语句的时候，需要特别注意这点。通常使用统一的编码规则是一个好的习惯，在这里这个好习惯会让你的系统运行得更快。

当查询语句中有一些不确定的数据时，则不会被缓存。例如包含函数 NOW() 或者 CURRENT_DATE() 的查询不会被缓存。类似的，包含 CURRENT_USER 或者 CONNECTION_ID() 的查询语句因为会根据不同的用户返回不同的结果，所以也不会被缓存。事实上，如果查询中
317 包含任何用户自定义函数、存储函数、用户变量、临时表、mysql 库中的系统表，或者任何包含列级别权限的表，都不会被缓存。（如果想知道所有情况，建议阅读 MySQL 官方手册。）

我们常听到："如果查询中包含一个不确定的函数，MySQL 则不会检查查询缓存"。这个说法是不正确的。因为在检查查询缓存的时候，还没有解析 SQL 语句，所以 MySQL 并不知道查询语句中是否包含这类函数。在检查查询缓存之前，MySQL 只做一件事情，就是通过一个大小写不敏感的检查看看 SQL 语句是不是以 SEL 开头。

准确的说法应该是："如果查询语句中包含任何的不确定函数，那么在查询缓存中是不可能找到缓存结果的"。因为即使之前刚刚执行了这样的查询，结果也不会放在查询缓存中。MySQL 在任何时候只要发现不能被缓存的部分，就会禁止这个查询被缓存。

所以，如果希望换成一个带日期的查询，那么最好将日期提前计算好，而不要直接使用函数。例如：

```
... DATE_SUB(CURRENT_DATE, INTERVAL 1 DAY) -- Not cacheable!
... DATE_SUB('2007-07-14', INTERVAL 1 DAY) -- Cacheable
```

因为查询缓存是在完整的 SELECT 语句基础上的，而且只是在刚收到 SQL 语句的时候才检查，所以子查询和存储过程都没办法使用查询缓存。在 MySQL 5.1 之前的版本中，绑定变量也无法使用查询缓存。

MySQL 的查询缓存在很多时候可以提升查询性能，在使用的时候，有一些问题需要特别注意。首先，打开查询缓存对读和写操作都会带来额外的消耗：

注 18： 对于这个规则，Percona Server 是个例外。它会先将所有的注释语句删除，然后再比较查询语句是否有缓存。这是一个通用的需求，这样可以在查询语句中带入更多的处理过程信息。前面第 3 章我们介绍的 MySQL 监控系统就依赖于此。

- 读查询在开始之前必须先检查是否命中缓存。
- 如果这个读查询可以被缓存，那么当完成执行后，MySQL 若发现查询缓存中没有这个查询，会将其结果存入查询缓存，这会带来额外的系统消耗。
- 这对写操作也会有影响，因为当向某个表写入数据的时候，MySQL 必须将对应表的所有缓存都设置失效。如果查询缓存非常大或者碎片很多，这个操作就可能会带来很大系统消耗（设置了很多的内存给查询缓存用的时候）。

虽然如此，查询缓存仍然可能给系统带来性能提升。但是，如上所述，这些额外消耗也可能不断增加，再加上对查询缓存操作是一个加锁排他操作，这个消耗可能不容小觑。

对 InnoDB 用户来说，事务的一些特性会限制查询缓存的使用。当一个语句在事务中修改了某个表，MySQL 会将这个表的对应的查询缓存都设置失效，而事实上，InnoDB 的多版本特性会暂时将这个修改对其他事务屏蔽。在这个事务提交之前，这个表的相关查询是无法被缓存的，所以所有在这个表上面的查询——内部或外部的事务——都只能在 318
该事务提交后才被缓存。因此，长时间运行的事务，会大大降低查询缓存的命中率。

如果查询缓存使用了很大量的内存，缓存失效操作就可能成为一个非常严重的问题瓶颈。如果缓存中存放了大量的查询结果，那么缓存失效操作时整个系统都可能会僵死一会儿。因为这个操作是靠一个全局锁操作保护的，所有需要做该操作的查询都要等待这个锁，而且无论是检测是否命中缓存、还是缓存失效检测都需要等待这个全局锁。第 3 章中有一个真实的案例，为大家展示查询缓存过大时带来的系统消耗。

7.12.2 查询缓存如何使用内存

查询缓存是完全存储在内存中的，所以在配置和使用它之前，我们需要先了解它是如何使用内存的。除了查询结果之外，需要缓存的还有很多别的维护相关的数据。这和文件系统有些类似：需要一些内存专门用来确定哪些内存目前是可用的、哪些是已经用掉的、哪些用来存储数据表和查询结果之前的映射、哪些用来存储查询字符串和查询结果。

这些基本的管理维护数据结构大概需要 40KB 的内存资源，除此之外，MySQL 用于查询缓存的内存被分成一个个的数据块，数据块是变长的。每一个数据块中，存储了自己的类型、大小和存储的数据本身，还外加指向前一个和后一个数据块的指针。数据块的类型有：存储查询结果、存储查询和数据表的映射、存储查询文本，等等。不同的存储块，在内存使用上并没有什么不同，从用户角度来看无须区分它们。

当服务器启动的时候，它先初始化查询缓存需要的内存。这个内存池初始是一个完整的空闲块。这个空闲块的大小就是你所配置的查询缓存大小再减去用于维护元数据的数据结构所消耗的空间。

当有查询结果需要缓存的时候，MySQL 先从大的空间块中申请一个数据块用于存储结果。这个数据块需要大于参数 query_cache_min_res_unit 的配置，即使查询结果远远小于此，仍需要至少申请 query_cache_min_res_unit 空间。因为需要在查询开始返回结果的时候就分配空间，而此时是无法预知查询结果到底多大的，所以 MySQL 无法为每一个查询结果精确分配大小恰好匹配的缓存空间。

因为需要先锁住空间块，然后找到合适大小数据块，所以相对来说，分配内存块是一个非常慢的操作。MySQL 尽量避免这个操作的次数。当需要缓存一个查询结果的时候，它先选择一个尽可能小的内存块（也可能选择较大的，这里将不介绍细节），然后将结
319 果存入其中。如果数据块全部用完，但仍有剩余数据需要存储，那么 MySQL 会申请一块新数据块——仍然是尽可能小的数据块——继续存储结果数据。当查询完成时，如果申请的内存空间还有剩余，MySQL 会将其释放，并放入空闲内存部分。图 7-3 展示了这个过程[注19]。

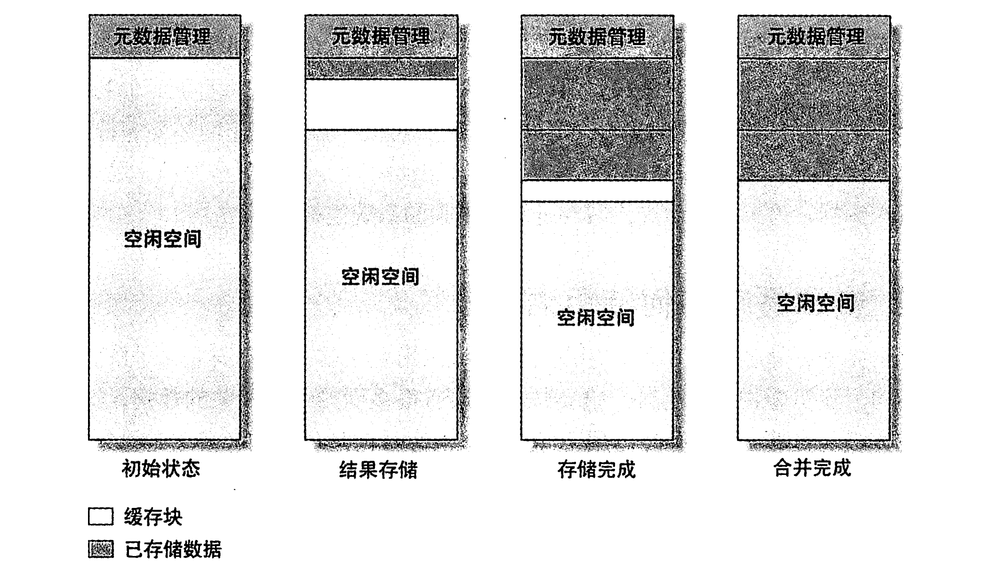


图7-3：查询缓存如何分配内存来存储结果数据

我们上面说的“分配内存块”，并不是指通过函数 malloc() 向操作系统申请内存，这个操作只在初次创建查询缓存的时候执行一次。这里“分配内存块”是指在空闲块列表中找到一个合适的内存块，或者从正在使用的、待淘汰的内存块中回收再使用。也就是说，

注 19： 这里绘制的查询缓存内存分配图，仍然是一种简化的情况。MySQL 实际管理查询缓存的方式比这要更复杂。如果你想知道更多的细节，在源代码文件 *sql/sql_cache.cc* 开头的注释中有非常详细的解释。

这里 MySQL 自己管理一大块内存，而不依赖操作系统的内存管理。

至此，一切都看起来很简单。不过实际情况比图 7-3 要更复杂。例如，我们假设平均查询结果非常小，服务器在并发地向不同的两个连接返回结果，返回完结果后 MySQL 回收剩余数据块空间时会发现，回收的数据块小于 query_cache_min_res_unit，所以不能够直接在后续的内存块分配中使用。如果考虑到这种情况，数据块的分配就更复杂些，如图 7-4 所示。

320

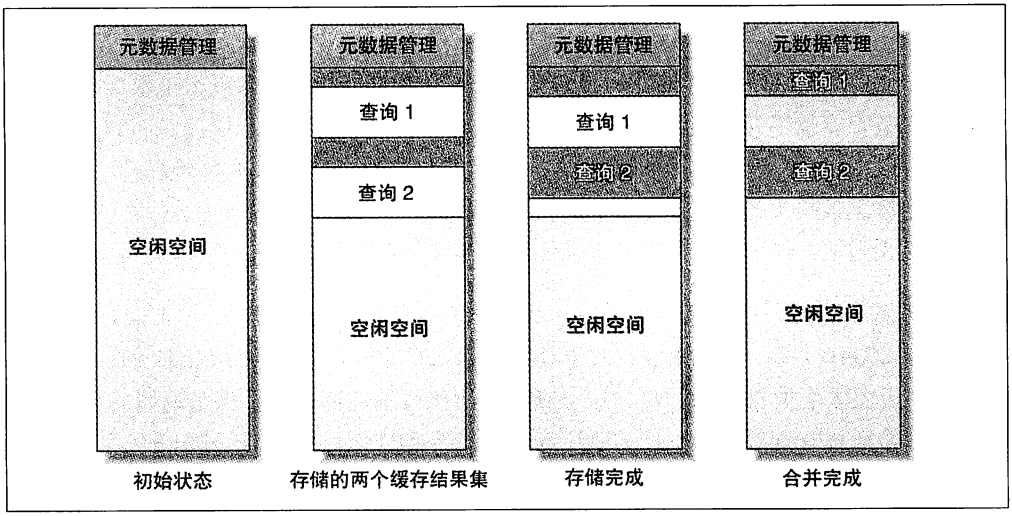


图7-4：查询缓存中存储查询结果后剩余的碎片

在收缩第一个查询结果使用的缓存空间时，就会在第二个查询结果之间留下一个“空隙”——一个非常小的空闲空间，因为小于 query_cache_min_res_unit 而不能再次被查询缓存使用。这类“空隙”我们称为“碎片”，这在内存管理、文件系统管理上都是经典问题。有很多种情况都会导致碎片，例如缓存失效时，可能导致留下太小的数据块无法在后续缓存中使用。

7.12.3 什么情况下查询缓存能发挥作用

并不是什么情况下查询缓存都会提高系统性能的。缓存和失效都会带来额外的消耗，所以只有当缓存带来的资源节约大于其本身的资源消耗时才会给系统带来性能提升。这跟具体的服务器压力模型有关。

理论上，可以通过观察打开或者关闭查询缓存时候的系统效率来决定是否需要开启查询缓存。关闭查询缓存时,每个查询都需要完整的执行,每一次写操作执行完成后立刻返回；打开查询缓存时，每次读请求先检查缓存是否命中，如果命中则立刻返回，否则就完整地执行查询，每次写操作则需要检查查询缓存中是否有需要失效的缓存，然后再返回。

这个过程还比较简单明了，但是要评估打开查询缓存是否能够带来性能提升却并不容易。还有一些外部的因素需要考虑，例如，查询缓存可以降低查询执行的时间，但是却不能减少查询结果传输的网络消耗，如果这个消耗是系统的主要瓶颈，那么查询缓存的作用也很小。

321 因为 MySQL 在 SHOW STATUS 中只能提供一个全局的性能指标，所以很难根据此来判断查询缓存是否能够提升性能[注20]。很多时候，全局平均不能反映实际情况。例如，打开查询缓存可以使得一个很慢的查询变得非常快，但是也会让其他查询稍微慢一点点。有时候如果能够让某些关键的查询速度更快，稍微降低一下其他查询的速度是值得的。不过，这种情况我们推荐使用 SQL_CACHE 来优化对查询缓存的使用。

对于那些需要消耗大量资源的查询通常都是非常适合缓存的。例如一些汇总计算查询，具体的如 COUNT() 等。总地来说，对于复杂的 SELECT 语句都可以使用查询缓存，例如多表 JOIN 后还需要做排序和分页，这类查询每次执行消耗都很大，但是返回的结果集却很小，非常适合查询缓存。不过需要注意的是，涉及的表上 UPDATE、DELETE 和 INSERT 操作相比 SELECT 来说要非常少才行。

一个判断查询缓存是否有效的直接数据是命中率，就是使用查询缓存返回结果占总查询的比率。当 MySQL 接收到一个 SELECT 查询的时候，要么增加 Qcache_hits 的值，要么增加 Com_select 的值。所以查询缓存命中率可以由如下公式计算：Qcache_hits/(Qcache_hits+Com_select)。

不过，查询缓存命中率是一个很难判断的数值。命中率多大才是好的命中率？具体情况要具体分析。只要查询缓存带来的效率提升大于查询缓存带来的额外消耗，即使 30% 命中率对系统性能提升也有很大好处。另外，缓存了哪些查询也很重要，例如，被缓存的查询本身消耗非常巨大，那么即使缓存命中率非常低，也仍然会对系统性能提升有好处。所以，没有一个简单的规则可以判断查询缓存是否对系统有好处。

任何 SELECT 语句没有从查询缓存中返回都称为“缓存未命中”。缓存未命中可能有如下几种原因：

- 查询语句无法被缓存，可能是因为查询中包含一个不确定的函数（如 CURRENT_

注 20： Percona 和 MariaDB 对 MySQL 慢日志进行了改进，会记录慢日志中的查询是否命中查询缓存。

DATA)，或者查询结果太大而无法缓存。这都会导致状态值 Qcache_not_cached 增加。

- MySQL 从未处理这个查询，所以结果也从不曾被缓存过。
- 还有一种情况是虽然之前缓存了查询结果，但是由于查询缓存的内存用完了，MySQL 需要将某些缓存"逐出"，或者由于数据表被修改导致缓存失效。（后续会详细介绍缓存失效。） 322

如果你的服务器上有大量缓存未命中，但是实际上绝大数查询都被缓存了，那么一定是有如下情况发生：

- 查询缓存还没有完成预热。也就是说，MySQL 还没有机会将查询结果都缓存起来。
- 查询语句之前从未执行过。如果你的应用程序不会重复执行一条查询语句，那么即使完成预热仍然会有很多缓存未命中。
- 缓存失效操作太多了。

缓存碎片、内存不足、数据修改都会造成缓存失效。如果配置了足够的缓存空间，而且 query_cache_min_res_unit 设置也合理的话，那么缓存失效应该主要是数据修改导致的。可以通过参数 Com_* 来查看数据修改的情况（包括 Com_update，Com_delete，等等），还可以通过 Qcache_lowmem_prunes 来查看有多少次失效是由于内存不足导致的。

在考虑缓存命中率的同时，通常还需要考虑缓存失效带来的额外消耗。一个极端的办法是，对某一个表先做一次只有查询的测试，并且所有的查询都命中缓存，而另一个相同的表则只做修改操作。这时，查询缓存的命中率就是 100%。但因为会给更新操作带来额外的消耗，所以查询缓存并不一定会带来总体效率的提升。这里，所有的更新语句都会做一次缓存失效检查，而检查的结果都是相同的，这会给系统带来额外的资源浪费。所以，如果你只是观察查询缓存的命中率的话，可能完全不会发现这样的问题。

在 MySQL 中如果更新操作和带缓存的读操作混合，那么查询缓存带来的好处通常很难衡量。更新操作会不断地使得缓存失效，而同时每次查询还会向缓存中再写入新的数据。所以只有当后续的查询能够在缓存失效前使用缓存才会有效地利用查询缓存。

如果缓存的结果在失效前没有被任何其他的 SELECT 语句使用，那么这次缓存操作就是浪费时间和内存。我们可以通过查看 Com_select 和 Qcache_inserts 的相对值来看看是否一直有这种情况发生。如果每次查询操作都是缓存未命中，然后需要将查询结果放到缓存中，那么 Qcache_inserts 的大小应该和 Com_select 相当。所以在缓存完成预热后，我们总希望看到 Qcache_inserts 远远小于 Com_select。不过由于缓存和服务器内部的复杂和多样性，仍然很难说，这个比率是多少才是一个合适的值。

323 所以，上面的“命中率”和“INSERTS 和 SELECT 比率”都无法直观地反应查询缓存的效率。那么还有什么直观的办法能够反映查询缓存是否对系统有好处？这里推荐查看另一个指标：“命中和写入”的比率，即 Qcache_hits 和 Qcache_inserts 的比值。根据经验来看，当这个比值大于 3 : 1 时通常查询缓存是有效的，不过这个比率最好能够达到 10 : 1。如果你的应用没有达到这个比率，那么就可以考虑禁用查询缓存了，除非你能够通过精确的计算得知：命中带来的性能提升大于缓存失效的消耗，并且查询缓存并没有成为系统的瓶颈。

每一个应用程序都会有一个“最大缓存空间”，甚至对一些纯读的应用来说也一样。最大缓存空间是能够缓存所有可能查询结果的缓存空间总和。理论上，对多数应用来说，这个数值都会非常大。而实际上，由于缓存失效的原因，大多数应用最后使用的缓存空间都比预想的要小。即使你配置了足够大的缓存空间，由于不断地失效，导致缓存空间一直都不会接近“最大缓存空间”。

通常可以通过观察查询缓存内存的实际使用情况，来确定是否需要缩小或者扩大查询缓存。如果查询缓存空间长时间都有剩余，那么建议缩小；如果经常由于空间不足而导致查询缓存失效，那么则需要增大查询缓存。不过需要注意，如果查询缓存达到了几十兆这样的数量级，是有潜在危险的。（这和硬件以及系统压力大小有关）。

另外，可能还需要和系统的其他缓存一起考虑，例如 InnoDB 的缓存池，或者 MyISAM 的索引缓存。关于这点是没法简单给出一个公式或者比率来判断的，因为真正的平衡点与应用程序有很大的关系。

最好的判断查询缓存是否有效的办法还是通过查看某类查询时间消耗是否增大或者减少来判断。Percona Server 通过扩展慢查询可以观察到一个查询是否命中缓存。如果查询缓存没有为系统节省时间，那么最好禁用它。

7.12.4 如何配置和维护查询缓存

一旦理解查询缓存工作的原理，配置起来就很容易了。它也只有很少的参数可供配置，如下所示。

query_cache_type

是否打开查询缓存。可以设置成 OFF、ON 或 DEMAND。DEMAND 表示只有在查询语句中明确写明 SQL_CACHE 的语句才放入查询缓存。这个变量可以是会话级别的也可以是全局级别的（会话级别和全局级别的概念请参考第 8 章）。

324 query_cache_size

查询缓存使用的总内存空间，单位是字节。这个值必须是 1 024 的整数倍，否则

MySQL 实际分配的数据会和你指定的略有不同。

query_cache_min_res_unit

在查询缓存中分配内存块时的最小单位。在前面我们已经介绍了这个参数，后面我们还将进一步讨论它。

query_cache_limit

MySQL 能够缓存的最大查询结果。如果查询结果大于这个值，则不会被缓存。因为查询缓存在数据生成的时候就开始尝试缓存数据，所以只有当结果全部返回后，MySQL 才知道查询结果是否超出限制。

如果超出，MySQL 则增加状态值 Qcache_not_cached，并将结果从查询缓存中删除。如果你事先知道有很多这样的情况发生，那么建议在查询语句中加入 SQL_NO_CACHE 来避免查询缓存带来的额外消耗。

query_cache_wlock_invalidate

如果某个数据表被其他的连接锁住，是否仍然从查询缓存中返回结果。这个参数默认是 OFF，这可能在一定程序上会改变服务器的行为，因为这使得数据库可能返回其他线程锁住的数据。将参数设置成 ON，则不会从缓存中读取这类数据，但是这可能会增加锁等待。对于绝大数应用来说无须注意这个细节，所以默认设置通常是没有问题的。

配置查询缓存通常很简单，但是如果想知道修改这些参数会带来哪些改变，则是一项很复杂的工作。后续的章节，我们将帮助你来决定怎样设置这些参数。

减少碎片

没什么办法能够完全避免碎片，但是选择合适的 query_cache_min_res_unit 可以帮你减少由碎片导致的内存空间浪费。设置合适的值可以平衡每个数据块的大小和每次存储结果时内存块申请的次数。这个值太小，则浪费的空间更少，但是会导致更频繁的内存块申请操作；如果这个值设置得太大，那么碎片会很多。调整合适的值其实是在平衡内存浪费和 CPU 消耗。

这个参数的最合适的大小和应用程序的查询结果的平均大小直接相关。可以通过内存实际消耗（query_cache_size-Qcache_free_memory）除以 Qcache_queries_in_cache 计算单个查询的平均缓存大小。如果你的应用程序的查询结果很不均匀，有的结果很大，有的结果很小，那么碎片和反复的内存块分配可能无法避免。如果你发现缓存一个非常大的结果并没有什么意义（通常确实是这样），那么你可以通过参数 query_cache_limit 限制可以缓存的最大查询结果，借此大大减少大的查询结果的缓存，最终减少内存碎片的发生。 325

还可以通过参数 Qcache_free_blocks 来观察碎片。参数 Qcache_free_blocks 反映了查询缓存中空闲块的多少，在图 7-4 的配置中我们看到，有两个空闲块。最糟糕的情况是，任何两个存储结果的数据块之间都有一个非常小的空闲块。所以如果 Qcache_free_blocks 大小恰好达到 Qcache_total_blocks/2，那么查询缓存就有严重的碎片问题。而如果你还有很多空闲块，而状态值 Qcache_lowmem_prunes 还不断地增加，则说明由于碎片导致了过早地在删除查询缓存结果。

可以使用命令 FLUSH QUERY CACHE 完成碎片整理。这个命令会将所有的查询缓存重新排序，并将所有的空闲空间都聚集到查询缓存的一块区域上。不过需要注意，这个命令并不会将查询缓存清空，清空缓存由命令 RESET QUERY CACHE 完成。FLUSH QUERY CACHE 会访问所有的查询缓存，在这期间任何其他的连接都无法访问查询缓存，从而会导致服务器僵死一段时间，使用这个命令的时候需要特别小心这点。另外，根据经验，建议保持查询缓存空间足够小，以便在维护时可以将服务器僵死控制在非常短的时间内。

提高查询缓存的使用率

如果查询缓存不再有碎片问题，但你仍然发现命中率很低，还可能是查询缓存的内存空间太小导致的。如果 MySQL 无法为一个新的查询缓存结果的时候，则会选择删除某个老的缓存结果。

当由于这个原因导致删除老的缓存结果时，会增加状态值 Qcache_lowmem_prunes。如果这个值增加得很快，那么可能是由下面两个原因导致的：

- 如果还有很多空闲块，那么碎片可能是罪魁祸首（参考前面的小节）。
- 如果这时没什么空闲块了，就说明在这个系统压力下，你分配的查询缓存空间不够大。你可以通过检查状态值 Qcache_free_memory 来查看还有多少没有使用的内存。

如果空闲块很多，碎片很少，也没有什么由于内存导致的缓存失效，但是命中率仍然很低，那么很可能说明，在你的系统压力下，查询缓存并没有什么好处。一定是什么原因导致查询缓存无法为系统服务，例如有大量的更新或者查询语句本身都不能被缓存。

如果在观察命中率时，仍然无法确定查询缓存是否给系统带来了好处，那么可以通过禁用它，然后观察系统的性能，再重新打开它，观察性能变化，据此来判断查询缓存是否给系统带来了好处。可以通过将 query_cache_size 设置成 0，来关闭查询缓存。（改变 query_cache_type 的全局值并不会影响已经打开的连接，也不会将查询缓存的内存释放
326 给系统。）你还可以通过系统测试来验证，不过一般都很难精确地模拟实际情况。

图 7-5 展示了一个用来分析和配置查询缓存的流程图。

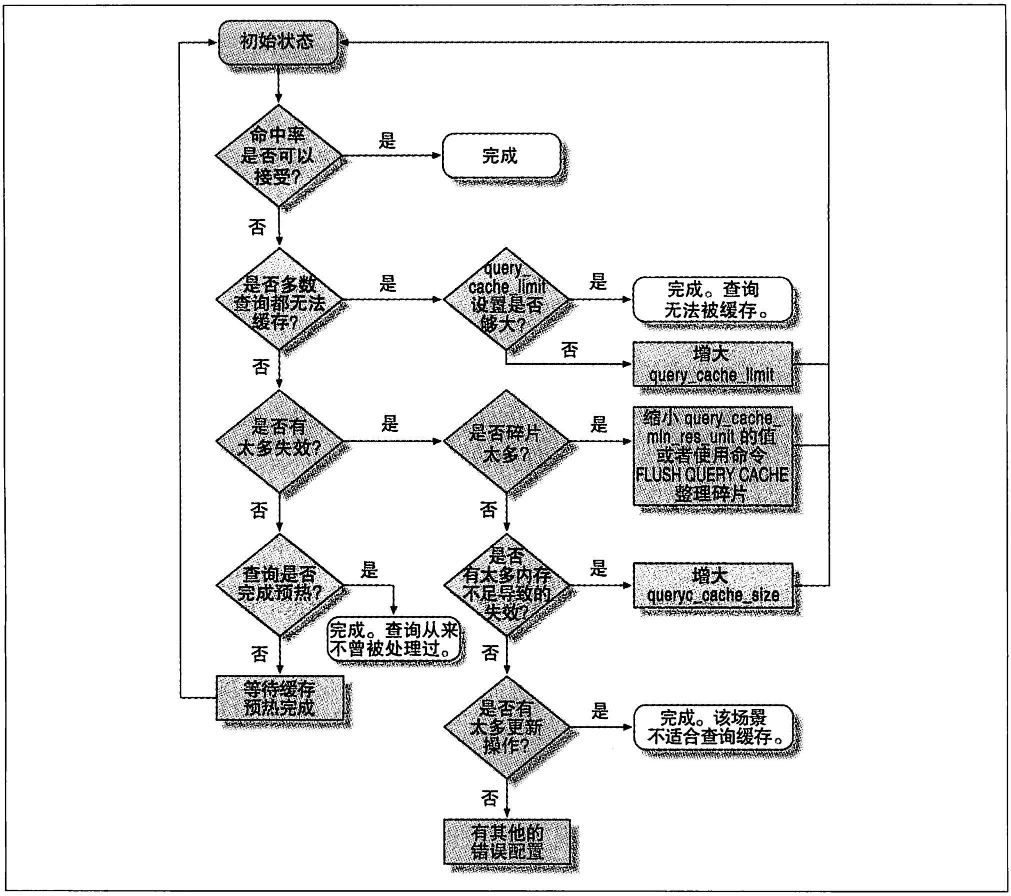


图7-5：如何分析和配置查询缓存

7.12.5 InnoDB 和查询缓存

因为 InnoDB 有自己的 MVCC 机制，所以相比其他存储引擎，InnoDB 和查询缓存的交互要更加复杂。MySQL 4.0 版本中，在事务处理中查询缓存是被禁用的，从 4.1 和更新的 InnoDB 版本开始，InnoDB 会控制在一个事务中是否可以使用查询缓存，InnoDB 会同时控制对查询缓存的读（从缓存中获取查询结果）和写操作（向查询缓存写入结果）。

事务是否可以访问查询缓存取决于当前事务 ID，以及对应的数据表上是否有锁。每一个 InnoDB 表的内存数据字典都保存了一个事物 ID 号，如果当前事务 ID 小于该事务 ID，则无法访问查询缓存。 327

如果表上有任何的锁，那么对这个表的任何查询语句都是无法被缓存的。例如，某个事务执行了 `SELECT FOR UPDATE` 语句，那么在这个锁释放之前，任何其他的事务都无法从

查询缓存中读取与这个表相关的缓存结果。

当事务提交时，InnoDB 持有锁，并使用当前的一个系统事务 ID 更新当前表的计数器。锁一定程度上说明事务需要对表进行修改操作，当然有可能事务获得锁，却不进行任何更新操作，但是如果想更新任何表的内容，获得相应锁则是前提条件。InnoDB 将每个表的计数器设置成某个事务 ID，而这个事务 ID 就代表了当前存在的且修改了该表的最大的事务 ID。

那么下面的一些事实也就成立：

- 所有大于该表计数器的事务才可以使用查询缓存。例如当前系统的事务 ID 是 5，且事务获取了该表的某些记录的锁，然后进行事务提交操作，那么事务 1 至 4，都不应该再读取或者向查询缓存写入任何相关的数据。
- 该表的计数器并不是直接更新为对该表进行加锁的事务 ID，而是被更新成一个系统事务 ID。所以，会发现该事务自身后续的更新操作也无法读取和修改查询缓存。

查询缓存存储、检索和失效操作都是在 MySQL 层面完成，InnoDB 无法绕过或者延迟这个行为。但 InnoDB 可以在事务中显式地告诉 MySQL 何时应该让某个表的查询缓存都失效。在有外键限制的时候这是必须的，例如某个 SQL 语句有 `ON DELETE CASCADE`，那么相关联表的查询缓存也是要一起失效的。

原则上，在 InnoDB 的 MVCC 架构下，当某些修改不影响其他事务读取一致的数据时，是可以使用查询缓存的。但是这样实现起来会非常复杂，InnoDB 做了一个简化，让所有有加锁操作的事务都不使用任何查询缓存，这个限制其实并不是必须的。

7.12.6 通用查询缓存优化

库表结构的设计、查询语句、应用程序设计都可能会影响到查询缓存的效率。除了前文介绍的之外，这里还有一些要点需要注意：

328
- 用多个小表代替一个大表对查询缓存有好处。这个设计将会使得失效策略能够在一个更合适的粒度上进行。当然，不要让这个原则过分影响你的设计，毕竟其他的一些优势可能很容易就弥补了这个问题。
- 批量写入时只需要做一次缓存失效，所以相比单条写入效率更好。（另外需要注意，不要同时做延迟写和批量写，否则可能会因为失效导致服务器僵死较长时间。）
- 因为缓存空间太大，在过期操作的时候可能会导致服务器僵死。一个简单的解决办法就是控制缓存空间的大小（`query_cache_size`），或者直接禁用查询缓存。
- 无法在数据库或者表级别控制查询缓存，但是可以通过 `SQL_CACHE` 和 `SQL_NO_CACHE` 来控制某个 `SELECT` 语句是否需要进行缓存。你还可以通过修改会话级别的变量

query_cache_type 来控制查询缓存。

- 对于写密集型的应用来说，直接禁用查询缓存可能会提高系统的性能。关闭查询缓存可以移除所有相关的消耗。例如将 query_cache_size 设置成 0，那么至少这部分就不再消耗任何内存了。
- 因为对互斥信号量的竞争，有时直接关闭查询缓存对读密集型的应用也会有好处。如果你希望提高系统的并发，那么最好做一个相关的测试，对比打开和关闭查询缓存时候的性能差异。

如果不想所有的查询都进入查询缓存，但是又希望某些查询走查询缓存，那么可以将 query_cache_type 设置成 DEMAND，然后在希望缓存的查询中加上 SQL_CACHE。这虽然需要在查询中加入一些额外的语法，但是可以让你非常自由地控制哪些查询需要被缓存。相反，如果希望缓存多数查询，而少数查询又不希望缓存，那么你可以使用关键字 SQL_NO_CACHE。

7.12.7 查询缓存的替代方案

MySQL 查询缓存工作的原则是：执行查询最快的方式就是不去执行，但是查询仍然需要发送到服务器端，服务器也还需要做一点点工作。如果对于某些查询完全不需要与服务器通信效果会如何呢？这时客户端的缓存可以很大程度上帮你分担 MySQL 服务器的压力。我们将在第 14 章详细介绍更多关于缓存的内容。

7.13 总结

329

本章详细介绍了前面各个章节中提到的一些 MySQL 特性。这里我们将再来回顾一下其中的一些重点内容。

分区表

分区表是一种粗粒度的、简易的索引策略，适用于大数据量的过滤场景。最适合的场景是，在没有合适的索引时，对其中几个分区进行全表扫描，或者是只有一个分区和索引是热点，而且这个分区和索引能够都在内存中；限制单表分区数不要超过 150 个，并且注意某些导致无法做分区过滤的细节，分区表对于单条记录的查询并没有什么优势，需要注意这类查询的性能。

视图

对好几个表的复杂查询，使用视图有时候会大大简化问题。当视图使用临时表时，无法将 WHERE 条件下推到各个具体的表，也不能使用任何索引，需要特别注意这类查询的性能。如果为了便利，使用视图是很合适的。

外键

外键限制会将约束放到 MySQL 中，这对于必须维护外键的场景，性能会更高。不过这也会带来额外的复杂性和额外的索引消耗，还会增加多表之间的交互，会导致系统中更多的锁和竞争。外键可以被看作是一个确保系统完整性的额外的特性，但是如果设计的是一个高性能的系统，那么外键就显得很臃肿了。很多人在更在意系统的性能的时候都不会使用外键，而是通过应用程序来维护。

存储过程

MySQL 本身实现了存储过程、触发器、存储函数和事件，老实说，这些特性并没什么特别的。而且对于基于语句的复制还有很多问题。通常，使用这些特性可以帮你节省很多的网络开销——很多情况下，减少网络开销可以大大提升系统的性能。在某些经典的场景下你可以使用这些特性(例如中心化业务逻辑、绕过权限系统，等等)，但需要注意在 MySQL 中，这些特性并没有别的数据库系统那么成熟和全面。

绑定变量

当查询语句的解析和执行计划生成消耗了主要的时间，那么绑定变量可以在一定程度上解决问题。因为只需要解析一次，对于大量重复类型的查询语句，性能会有很大的提高。另外，执行计划的缓存和传输使用的二进制协议，这都使得绑定变量的方式比普通 SQL 语句执行的方式要更快。

330 插件

使用 C 或者 C++ 编写的插件可以让你最大程度地扩展 MySQL 功能。插件功能非常强大，我们已经编写了很多 UDF 和插件，在 MySQL 中解决了很多问题。

字符集

字符集是一种字节到字符之间的映射，而校对规则是指一个字符集的排序方法。很多人都使用 Latin1（默认字符集，对英语和某些欧洲语言有效）或者 UTF-8。如果使用的是 UTF-8，那么在使用临时表和缓冲区的时候需要注意：MySQL 会按照每个字符三个字节的最大占用空间来分配存储空间，这可能消耗更多的内存或者磁盘空间。注意让字符集和 MySQL 字符集配置相符，否则可能会由于字符集转换让某些索引无法正常使用。

全文索引

在本书编写的时候只有 MyISAM 支持全文索引，不过据说从 MySQL 5.6 开始，InnoDB 也将支持全文索引。MyISAM 因为在锁粒度和崩溃恢复上的缺点，使得在大型全文索引场景中基本无法使用。这时，我们通常帮助客户构建和使用 Sphinx 来解决全文索引的问题。

XA 事务

很少有人用 MySQL 的 XA 事务特性。除非你真正明白参数 `innodb_support_xa` 的意义，否则不要修改这个参数的值，并不是只有显式使用 XA 事务时才需要设置这

个参数。InnoDB 和二进制日志也是需要使用 XA 事务来做协调的，从而确保在系统崩溃的时候，数据能够一致地恢复。

查询缓存

完全相同的查询在重复执行的时候，查询缓存可以立即返回结果，而无须在数据库中重新执行一次。根据我们的经验，在高并发压力环境中查询缓存会导致系统性能的下降，甚至僵死。如果你一定要使用查询缓存，那么不要设置太大内存，而且只有在明确收益的时候才使用。那该如何判断是否应该使用查询缓存呢？建议使用 Percona Server，观察更细致的日志，并做一些简单的计算。还可以查看缓存命中率（并不总是有用）、"INSERTS 和 SELECT 比率"（这个参数也并不直观）、或者"命中和写入比率"（这个参考意义较大）。查询缓存是一个非常方便的缓存，对应用程序完全透明，无须任何额外的编码，但是，如果希望有更高的缓存效率，我们建议使用 *memcached* 或者其他类似的解决方案。第 14 章介绍了更多的细节供大家参考。

第 8 章 <331>

优化服务器设置

在这一章，我们将解释为 MySQL 服务器创建一个靠谱的配置文件的过程。这是一个很绕的过程，有很多有意思的关注点和值得关注的思路。关注这些点很有必要，因为创建一个好配置的最快方法不是从学习配置项开始，也不是从问哪个配置项应该怎么设置或者怎么修改开始，更不是从检查服务器行为和询问哪个配置项可以提升性能开始。最好是从理解 MySQL 内核和行为开始。然后可以利用这些知识来指导配置 MySQL。最后，可以将想要的配置和当前配置进行比较，然后纠正重要并且有价值的不同之处。

人们经常问，"我的服务器有 32GB 内存，12 核 CPU，怎样配置最好？" 很遗憾，问题没这么简单。服务器的配置应该符合它的工作负载、数据，以及应用需求，并不仅仅看硬件的情况。

MySQL 有大量可以修改的参数——但不应该随便去修改。通常只需要把基本的项配置正确（大部分情况下只有很少一些参数是真正重要的），应该将更多的时间花在 schema 的优化、索引，以及查询设计上。在正确地配置了 MySQL 的基本配置项之后，再花力气去修改其他配置项的收益通常就比较小了。

从另一方面来说，没用的配置导致潜在风险的可能更大。我们碰到过不止一个"高度调优"过的服务器不停地崩溃，停止服务或者运行缓慢，结果都是因为错误的配置导致的。我们将花一点时间来解释为什么会发生这种情况，并且告诉大家什么是不该做的。

那么什么是该做的呢？确保基本的配置是正确的，例如 InnoDB 的 Buffer Pool 和日志文件缓存大小，如果想防止出问题（提醒一下，这样做通常不能提升性能——它们只能避免问题），就设置一个比较安全和稳健的值，剩下的配置就不用管了。如果碰到了问题，可以使用第 3 章提到的技巧小心地进行诊断。如果问题是由于服务器的某部分导致的，而这恰好可以通过某个配置项解决，那么需要做的就是更改配置。

332 有时候，在某些特定的场景下，也有可能设置某些特殊的配置项会有显著的性能提升。但无论如何，这些特殊的配置项不应该成为服务器基本配置文件的一部分。只有当发现特定的性能问题才应该设置它们。这就是为什么我们不建议通过寻找有问题的地方修改配置项的原因。如果有些地方确实需要提升，也需要在查询响应时间上有所体现。最好是从查询语句和响应时间入手来开始分析问题，而不是通过配置项。这可以节省大量的时间，避免很多的问题。

另一个节省时间和避免麻烦的好办法是使用默认配置，除非是明确地知道默认值会有问题。很多人都是在默认配置下运行的，这种情况非常普遍。这使得默认配置是经过最多实际测试的。对配置项做一些不必要的修改可能会遇到一些意料之外的 bug。

8.1 MySQL 配置的工作原理

在讨论如何配置 MySQL 之前，我们先来解释一下 MySQL 的配置机制。MySQL 对配置要求非常宽松，但是下面这些建议可能会为你节省大量的工作和时间。

首先应该知道的是 MySQL 从哪里获得配置信息：命令行参数和配置文件。在类 UNIX 系统中，配置文件的位置一般在 */etc/my.cnf* 或者 */etc/mysql/my.cnf*。如果使用操作系统的启动脚本，这通常是唯一指定配置设置的地方。如果手动启动 MySQL，例如在测试安装时，也可以在命令行指定设置。实际上，服务器会读取配置文件的内容，删除所有注释和换行，然后和命令行选项一起处理。

> 关于术语的说明：因为很多 MySQL 命令行选项跟服务器变量相同，我们有时把选项和变量替换使用。大部分变量和它们对应的命令行选项名称一样，但是有一些例外。例如，*--memlock* 选项设置了 `locked_in_memory` 变量。

任何打算长期使用的设置都应该写到全局配置文件，而不是在命令行特别指定。否则，如果偶然在启动时忘了设置就会有风险。把所有的配置文件放在同一个地方以方便检查也是个好办法。

一定要清楚地知道服务器配置文件的位置！我们见过有些人尝试修改配置文件但是不生效，因为他们修改的并不是服务器读取的文件，例如 Debian 下，*/etc/mysql/my.cnf* 才是
333 MySQL 读取的配置文件，而不是 */etc/my.cnf*。有时候好几个地方都有配置文件，也许是因为之前的系统管理员也没搞清楚情况（因此在各个可能的位置都放了一份）。如果不知道当前使用的配置文件路径，可以尝试下面的操作：

```
$ which mysqld
/usr/sbin/mysqld
$ /usr/sbin/mysqld --verbose --help | grep -A 1 'Default options'
Default options are read from the following files in the given order:
/etc/mysql/my.cnf ~/.my.cnf /usr/etc/my.cnf
```

对于服务器上只有一个 MySQL 实例的典型安装，这个命令很有用。也可以设计更复杂的配置，但是没有标准的方法告诉你怎么来做。MySQL 发行版包含了一个现在废弃了的程序，叫 *mysqlmanager*，可以在一个有多个独立部分的配置文件上运行多个实例。（现在已经被一样古老的 *mysqld_multi* 脚本替代。）然而许多操作系统发行版本在启动脚本中并不包含或使用这个程序。实际上，很多系统甚至没有使用 MySQL 提供的启动脚本。

配置文件通常分成多个部分，每个部分的开头是一个用方括号括起来的分段名称。MySQL 程序通常读取跟它同名的分段部分，许多客户端程序还会读取 client 部分，这是一个存放公用设置的地方。服务器通常读取 mysqld 这一段。一定要确认配置项放在了文件正确的分段中，否则配置是不会生效的。

8.1.1 语法、作用域和动态性

配置项设置都使用小写，单词之间用下画线或横线隔开。下面的例子是等价的，并且可能在命令行和配置文件中都看到这两种格式：

```
/usr/sbin/mysqld --auto-increment-offset=5
/usr/sbin/mysqld --auto_increment_offset=5
```

我们建议使用一种固定的风格。这样在配置文件中搜索配置项时会容易得多。

配置项可以有多个作用域。有些设置是服务器级的（全局作用域），有些对每个连接是不同的（会话作用域），剩下的一些是对象级的。许多会话级变量跟全局变量相等，可以认为是默认值。如果改变会话级变量，它只影响改动的当前连接，当连接关闭时所有参数变更都会失效。下面有一些例子，你应该清楚这些不同类型的行为：

- query_cache_size 变量是全局的。
- sort_buffer_size 变量默认是全局相同的，但是每个线程里也可以设置。
- join_buffer_size 变量也有全局默认值且每个线程是可以设置的，但是若一个查询中关联多张表，可以为每个关联分配一个关联缓冲（join buffer），所以每个查询可能有多个关联缓冲。 334

另外，除了在配置文件中设置变量，有很多变量（但不是所有）也可以在服务器运行时修改。MySQL 把这些归为动态配置变量。下面的语句展示了动态改变 sort_buffer_size 的会话值和全局值的不同方式：

```
SET          sort_buffer_size  = <value>;
SET GLOBAL   sort_buffer_size  = <value>;
SET        @@sort_buffer_size := <value>;
SET @@session.sort_buffer_size := <value>;
SET  @@global.sort_buffer_size := <value>;
```

如果动态地设置变量，要注意 MySQL 关闭时可能丢失这些设置。如果想保持这些设置，还是需要修改配置文件。

如果在服务器运行时修改了变量的全局值，这个值对当前会话和其他任何已经存在的会话都不起效果，这是因为会话的变量值是在连接创建时从全局值初始化来的。在每次变更之后，应该检查 SHOW GLOBAL VARIABLES 的输出，确认已经按照期望变更了。

有些变量使用了不同的单位，所以必须知道每个变量的正确单位。例如，table_cache 变量指定了表可以被缓存的数量，而不是表可以被缓存的字节数。key_buffer_size 则是以字节为单位，还有一些其他变量指定的是页的数量或者其他单位，例如百分比。

许多变量可以通过后缀指定单位，例如 1M 表示一百万字节。然而，这只能在配置文件或者作为命令行参数时有效。当使用 SQL 的 SET 命令时，必须使用数字值 1048576，或者 1024*1024 这样的表达式。但在配置文件中不能使用表达式。

有个特殊的值可以通过 SET 命令赋值给变量：DEFAULT。把这个值赋给会话级变量可以把变量改为使用全局值，把它赋值给全局变量可以设置这个变量为编译内置的默认值（不是在配置文件中指定的值）。当需要重置会话级变量的值回到连接刚打开的时候，这是很有用的。建议不要对全局变量这么用，因为可能它做的事不是你希望的，它不会把值设置到服务器刚启动时候的那个状态。

335 8.1.2 设置变量的副作用

动态设置变量可能导致意外的副作用，例如从缓冲中刷新脏块。务必小心那些可以在线更改的设置，因为它们可能导致数据库做大量的工作。

有时可以通过名称推断一个变量的作用。例如，max_heap_table_size 的作用就像听起来那样：它指定隐式内存临时表最大允许的大小。然而，命名约定并不完全一样，所以不能总是通过看名称来猜测一个变量有什么效果。

让我们来看一些常用的变量和动态修改它们的效果。

key_buffer_size

设置这个变量可以一次性为键缓冲区（key buffer，也叫键缓存 key cache）分配所有指定的空间。然而，操作系统不会真的立刻分配内存，而是到使用时才真正分配。

例如设置键缓冲的大小为 1GB，并不意味着服务器立刻分配 1GB 的内存。（我们下一章会讨论如何查看服务器的内存使用。）

MySQL 允许创建多个键缓存，这一章后面我们会探讨这个问题。如果把非默认键缓存的这个变量设置为 0，MySQL 将丢弃缓存在该键缓存中的索引，转而使用默认键缓存，并且当不再有任何引用时会删除该键缓存。为一个不存在的键缓存设置这个变量，将会创建新的键缓存。对一个已经存在的键缓存设置非零值，会导致刷新该键缓存的内容。这会阻塞所有尝试访问该键缓存的操作，直到刷新操作完成。

table_cache_size

设置这个变量不会立即生效——会延迟到下次有线程打开表才有效果。当有线程打开表时，MySQL 会检查这个变量的值。如果值大于缓存中的表的数量，线程可以把最新打开的表放入缓存；如果值比缓存中的表数小，MySQL 将从缓存中删除不常使用的表。

thread_cache_size

设置这个变量不会立即生效——将在下次有连接被关闭时产生效果。当有连接被关闭时，MySQL 检查缓存中是否还有空间来缓存线程。如果有空间，则缓存该线程以备下次连接重用；如果没有空间，它将销毁该线程而不再缓存。在这个场景中，缓存中的线程数，以及线程缓存使用的内存，并不会立刻减少；只有在新的连接删除缓存中的一个线程并使用后才会减少。（MySQL 只在关闭连接时才在缓存中增加线程，只在创建新连接时才从缓存中删除线程。）

query_cache_size 336

MySQL 在启动的时候，一次性分配并且初始化这块内存。如果修改这个变量（即使设置为与当前一样的值），MySQL 会立刻删除所有缓存的查询，重新分配这片缓存到指定大小，并且重新初始化内存。这可能花费较长的时间，在完成初始化之前服务器都无法提供服务，因为 MySQL 是逐个清理缓存的查询，不是一次性全部删掉。

read_buffer_size

MySQL 只会在有查询需要使用时才会为该缓存分配内存，并且会一次性分配该参数指定大小的全部内存。

read_rnd_buffer_size

MySQL 只会在有查询需要使用时才会为该缓存分配内存，并且只会分配需要的内存大小而不是全部指定的大小。（max_read_rnd_buffer_size 这个名字更能表达这个变量实际的含义。）

sort_buffer_size

MySQL 只会在有查询需要做排序操作时才会为该缓存分配内存。然后，一旦需要排序，MySQL 就会立刻分配该参数指定大小的全部内存，而不管该排序是否需要这么大的内存。

我们在其他地方也对这些参数做过更多细节的说明，这里不是一个完整的列表。这里的目的只是简单地告诉大家，当修改一些常见的变量时，会有哪些期望的行为发生。

对于连接级别的设置，不要轻易地在全局级别增加它们的值，除非确认这样做是对的。有一些缓存会一次性分配指定大小的全部内存，而不管实际上是否需要这么大，所以一个很大的全局设置可能导致浪费大量内存。更好的办法是，当查询需要时在连接级别单独调大这些值。

最常见的例子是 sort_buffer_size，该参数控制排序操作的缓存大小，应该在配置文件里把它配置得小一些，然后在某些查询需要排序时，再在连接中把它调大。在分配内存后，MySQL 会执行一些初始化的工作。

另外，即使是非常小的排序操作，排序缓存也会分配全部大小的内存，所以如果把参数设置得超过平均排序需求太多，将会浪费很多内存，增加额外的内存分配开销。许多读者认为内存分配是一个很简单的操作，听到内存分配的代价可能会很吃惊。不需要深入很多技术细节就可以讲清楚为什么内存分配也是昂贵的操作，内存分配包括了地址空间的分配，这相对来说是比较昂贵的。特别在 Linux 上，内存分配根据大小使用多种开销不同的策略。

总的来说，设置很大的排序缓存代价可能非常高，所以除非确定必须要这么大，否则不要增加排序缓存的大小。

337 如果查询必须使用一个更大的排序缓存才能比较好地执行，可以在查询执行前增加 sort_buffer_size 的值，执行完成后恢复为 DEFAULT。

下面是一个实际的例子：

```
SET @@session.sort_buffer_size := <value>;
-- Execute the query...
SET @@session.sort_buffer_size := DEFAULT;
```

可以将类似的代码封装在函数中以方便使用。其他可以设置的单个连接级别的变量有 read_buffer_size、read_rnd_buffer_size、tmp_table_size、以及 myisam_sort_buffer_size（在修复表的操作中会用到）。

如果有需要也可以保存并还原原来的自定义值，可以像下面这样做：

```
SET @saved_<unique_variable_name> := @@session.sort_buffer_size;
SET @@session.sort_buffer_size := <value>;
-- Execute the query...
SET @@session.sort_buffer_size := @saved_<unique_variable_name>;
```

排序缓冲大小是关注的众多"调优"中一个设置。一些人似乎认为越大越好，我们甚至见过把这个变量设为 1GB 的。这可能导致服务器尝试分配太多内存而崩溃，或者为查询初始化排序缓存时消耗大量的 CPU，这不是什么出乎意料的事。从 MySQL 的 Bug 37359 可以看到有关于这个问题的细节。

不要把排序缓存大小放在太重要的位置。查询真的需要 128MB 的内存来排序 10 行数据然后返回给客户端吗？思考一下查询语句是什么类型的排序、多大的排序，首先考虑通过索引和 SQL 写法来避免排序（看第 5 章和第 6 章），这比调优排序缓存要快得多。并且应该仔细分析查询开销，看看排序是否是无论如何都需要重点关注的部分。第 3 章有一个例子，一个查询执行了一个排序，但是没有花很多排序时间。

8.1.3 入门

设置变量时请小心，并不是值越大就越好，而且如果设置的值太高，可能更容易导致问题：可能会由于内存不足导致服务器内存交换，或者超过地址空间。[注1]

应该始终通过监控来确认生产环境中变量的修改，是提高还是降低了服务器的整体性能。338
基准测试是不够的，因为基准测试不是真实的工作负载。如果不对服务器的性能进行实际的测量，可能性能降低了都没有发现。我们见过很多情况，有人修改了服务器的配置，并认为它提高了性能，其实服务器的整体性能恶化了，因为在一个星期或一天的不同时间，工作负载是不一样的。

如果你经常做笔记，在配置文件中写好注释，可能会节省自己（和同事）大量的工作。一个更好的主意是把配置文件置于版本控制之下。无论如何，这是一个很好的做法，因为它让你有机会撤销变更。要降低管理很多配置文件的复杂性，简单地创建一个从配置文件到中央版本控制库的符号链接。

在开始改变配置之前，应该优化查询和 schema，至少先做明显要做的事情，例如添加索引。如果先深入调整配置，然后修改了查询语句和 schema，也许需要回头再次评估配置。请记住，除非硬件、工作负载和数据是完全静态的，否则都可能需要重新检查配置文件。实际上，大部分人的服务器甚至在一天中都没有稳定的工作负载——意味着对上午来说"完美"的配置，下午就不对了！显然，追求传说中的"完美"配置是完全不切实际的。因此，没有必要榨干服务器的每一点性能，实际上，这种调优的时间投入产出是非常小的。我们建议在"足够好"的时候就可以停下了，除非有理由相信停下会导致放弃重大的性能提升的机会。

注 1：我们见过的一个常见的错误是，配置一台新服务器的内存是另一台已经存在的服务器的两倍，并且——使用旧服务器的配置作为基线——创建一份新的配置，只是简单地在旧服务器的配置上乘以 2。这不起作用。

8.1.4 通过基准测试迭代优化

你也许期望（或者相信自己会期望）通过建立一套基准测试方案，然后不断迭代地验证对配置项的修改来找到最佳配置方案。通常我们都不建议大家这么做。这需要做非常多的工作和研究，并且大部分情况下潜在的收益是非常小的，这可能导致巨大的时间浪费。而把时间花在检查备份、监控执行计划的变动之类的事情上，可能会更有意义。

即使更改一个选项后基准测试出现了提升，也无法知道长期运行后这个变更会有什么副作用。基准测试也不能衡量一切，或者没有运行足够长的时间来检测系统的长期稳定性，修改就可能导致如周期性性能抖动或者周期性的慢查询等问题。这是很难察觉到的。

339 有的时候我们运行某些组合的基准测试，来仔细验证或压测服务器的某些特定部分，使得我们可以更好地理解这些行为。一个很好的例子是，我们使用了很多年的一些基准测试，用来理解 InnoDB 的刷新行为，来寻找更好的刷新算法，以适应多种工作负载和多种硬件类型。我们经常测试各种各样的设置，来理解它们的影响以及怎么优化它们。但这不是一件简单的事——这可能会花费很多天甚至很多个星期——而且对大部分人来说这没有收益，因为服务器特定部分的认识局限往往会掩盖了其他问题。例如，有时我们发现，特定的设置项组合，在特定的边缘场景可能有更好的性能，但是在实际生产环境这些配置项并不真的合适，例如，浪费大量的内存，或者优化了吞吐量却忽略了崩溃恢复的影响。

如果必须这样做，我们建议在开始配置服务器之前，开发一个定制的基准测试包。你必须做这些事情来包含所有可能的工作负载，甚至包含一些边缘的场景，例如很庞大很复杂的查询语句。在实际的数据上重放工作负载通常是一个好办法。如果已经定位到了一个特定的问题点——例如一个查询语句运行很慢——也可以尝试专门优化这个点，但是可能不知道这会对其他查询有什么负面影响。

最好的办法是一次改变一个或两个变量，每次一点点，每次更改后运行基准测试，确保运行足够长的时间来确认性能是否稳定。有时结果可能会令你感到惊讶，可能把一个变量调大了一点，观察到性能提升，然后再调大一点，却发现性能大幅下降。如果变更后性能有隐患，可能是某些资源用得太多了，例如，为缓冲区分配太多内存、频繁地申请和释放内存。另外，可能导致 MySQL 和操作系统或硬件之间的不匹配。例如，我们发现 `sort_buffer_size` 的最佳值可能会被 CPU 缓存的工作方式影响，还有 `read_buffer_size` 需要服务器的预读和 I/O 子系统的配置相匹配。更大并不总是更好，还可能更糟糕。一些变量也依赖于一些其他的东西，这需要通过经验和对系统架构的理解来学习。

什么情况下进行基准测试是好的建议

对于前面提到不建议大多数人执行基准测试的情况也有例外的时候。我们有时会建议人们跑一些迭代基准测试，尽管通常跟“服务器调优”有不同的内容。这里有一些例子：

- 如果有一笔大的投资，如购买大量新的服务器，可以运行一下基准测试以了解硬件需求。（这里的上下文指是容量规划，不是服务器调优），我们特别喜欢对不同大小的 InnoDB 缓冲池进行基准测试，这有助于我们制定一个“内存曲线”，以展示真正需要多少内存，不同的内存容量如何影响存储系统的要求。
- 如果想了解 InnoDB 从崩溃中恢复需要多久时间，可以反复设置一个备库，故意让它崩溃，然后“测试” InnoDB 在重启中需要花费多久时间来做恢复。这里的背景是做高可用性的规划。
- 以读为主的应用程序，在慢查询日志中捕捉所有的查询（或者用 *pt-query-digest* 分析 TCP 流量）是个很好的主意，在服务器完全打开慢查询日志记录时，使用 *pt-log-player* 重放所有的慢查询，然后用 *pt-query-digest* 来分析输出报告。这可以观察在不同硬件、软件和服务器设置下，查询语句运行的情况。例如，我们曾经帮助客户评估迁移到更多的内存但硬盘更慢的服务器上的性能变化。大多数查询变得更快，但一些分析型查询语句变慢，因为它们是 I/O 密集型的。这个测试的上下文背景就是不同工作负载的比较。

8.2 什么不该做

在我们开始配置服务器之前，希望鼓励大家去避免一些我们已经发现有风险或有害的做法。警告：本节有些观点可能会让有些人不舒服！

首先，不要根据一些“比率”来调优。一个经典的按“比率”调优的经验法则是，键缓存的命中率应该高于某个百分比，如果命中率过低，则应该增加缓存的大小。这是非常错误的意见。无论别人怎么跟你说，缓存命中率跟缓存是否过大或过小没有关系。首先，命中率取决于工作负载——某些工作负载就是无法缓存的，不管缓存有多大——其次，缓存命中没有什么意义，我们将在后面解释原因。有时当缓存太小时，命中率比较低，增加缓存的大小确实可以提高命中率。然而，这只是个偶然情况，并不表示这与性能或适当的缓存大小有任何关系。

341 这种相关性，有时候看起来似乎真正的问题是，人们开始相信它们将永远是真的。Oracle DBA 很多年前就放弃了基于命中率的调优，我们希望 MySQL DBA 也能跟着走[注2]。我们更强烈地希望人们不要去写“调优脚本”，把这些危险的做法编写到一起，并教导成千上万的人这么做。这引出了我们第二个不该做的建议：不要使用调优脚本！有几个这样的可以在互联网上找到的脚本非常受欢迎，最好是忽略它们[注3]。

我们还建议避免调（tuning）这个词，我们在前面几段中使用这个词是有点随意的。我们更喜欢使用“配置（Configuration）”或“优化（Optimize）”来代替（只要这是你真正在做的，见第 3 章）。“调优”这个词，容易让人联想到一个缺乏纪律的新手对服务器进行微调，并观察发生了什么。我们建议上一节的练习最好留给那些正在研究服务器内核的人。“调优”服务器可能浪费大量的时间。

另外说一句，在互联网搜索如何配置并不总是一个好主意。在博客、论坛等地方[注4]都可能找到很多不好的建议。虽然许多专家在网上贡献了他们了解的东西，但并不总是能容易地分辨出哪些是正确的建议。我们也不能给出中肯的建议在哪里能找到真正的专家[注5]。但我们可以说，可信的、声誉好的 MySQL 服务供应商一般比简单的互联网搜索更安全，因为有好的客户才可能做出正确的事情。然而，即使是他们的意见，没有经过测试和理解就使用，也可能有危险，因为它可能对某种解决方案有了思维定势，跟你的思维不一样，可能用了一种你无法理解的方法。

最后，不要相信很流行的内存消耗公式——是的，就是 MySQL 崩溃时自身输出的那个内存消耗公式（我们这里就不再重复了）。这个公式已经很古老了，它并不可靠，甚至也不是一个理解 MySQL 在最差情况下需要使用多少内存的有用的办法。在互联网上可能还会看到这个公式的很多变种。即使在原公式上增加了更多原来没有考虑到的因素，还是有同样的缺陷。事实上不可能非常准确地把握 MySQL 内存消耗的上限。MySQL 不是一个完全严格控制内存分配的数据库服务器。这个结论可以非常简单地证明，登录到服务器，并执行一些大量消耗内存的查询：

```
mysql> SET @crash_me_1 := REPEAT('a', @@max_allowed_packet);
mysql> SET @crash_me_2 := REPEAT('a', @@max_allowed_packet);
342 # ... run a lot of these ...
mysql> SET @crash_me_1000000 := REPEAT('a', @@max_allowed_packet);
```

注 2： 如果你还是不相信“按比率调优”的方法是错误的，请阅读 Cary Millsap 的 *Optimizing Oracle Performance*（O'Reilly 出版）。他甚至为这个主题专门写了一个附录，提供了一个可以智能地产生任何你想要的命中率的工具，甚至不管系统正运行得多么糟糕都可以做到很好的命中率！当然，这一切的目的都是为了说明比率是多么无用。

注 3： 一个例外：我们维护了一个（好用的）免费的在线配置工具，在 *http://tools.percona.com*。是的，我们确实有倾向性。

注 4： 问：（坏的）查询是如何形成的？答：这需要去问那些杀死了坏查询的 DBA 是怎么回事，查询本身是不可能回击的。

注 5： Percona 当然认为在 Percona 邮件组能找到真正的专家，所以说他们不能中肯。——译者注

在一个循环中运行这些语句，每次都创建新的变量，最后服务器内存必然耗尽，然后系统崩溃！运行这个测试不需要任何特殊权限。

在本节我们试图说明的观点是，有时候我们在那些认为我们很傲慢的人面前变得不受欢迎，他们认为我们正在试图诋毁他人，把自己塑造成唯一的权威，或者觉得我们是在试图推销我们的服务。我们的目的不是利己。我们只是看到非常多很糟糕的建议，如果没有足够的经验，这看上去似乎还是合理的。另外我们重复这么多次说明这些糟糕的建议，因为我们认为揭穿一些神话是很重要的，并提醒我们的读者要小心他们信任的那些人的专业水准。我们还是尽量避免在这里继续说这些不好听的吧。

8.3 创建 MySQL 配置文件

正如我们在本章开头提到的，没有一个适合所有场景的“最佳配置文件”，比方说，对一台有 16 GB 内存和 12 块硬盘的 4 路 CPU 服务器，不会有一个相应的“最佳配置文件”。应该开发自己的配置，因为即使是一个好的起点，也依赖于具体是如何使用服务器的。

MySQL 编译的默认设置并不都是靠谱的，虽然其中大部分都比较合适。它们被设计成不要使用大量的资源，因为 MySQL 的使用目标是非常灵活的，它并没有假设自己是服务器上唯一的应用。默认情况下，MySQL 只是使用恰好足够的资源来启动，运行一些少量数据的简单查询。如果有超过几 MB 的数据，就一定会需要自己定制 MySQL 配置。

你可能会先从一个包含在 MySQL 发行版本中的示例配置文件开始，但这些示例配置有自己的问题。例如，它们有很多注释掉的设置，可能会诱使你认为应该选择一个值，并取消注释（这有点让人联想到 Apache 配置文件）。同时它们有很多乏味的注释，只是为了解释选项的含义，但这些解释并不总是通顺、完整甚至正确的，有些选项甚至并不适用于流行的操作系统！最后，这些示例相对于现代的硬件和工作负载，总是过时的。

MySQL 专家们关于如何解决这些问题多年来进行了许多对话，但这些问题依然存在。下面是我们的建议：不要使用这些文件作为（创建配置文件的）起点，也不要使用操作系统的安装包自带的配置文件。最好是从头开始。

这就是本章要做的事情。实际上 MySQL 的可配置性太强也可以说是个弱点，看起来好像需要花很多时间在配置上，其实大多数配置的默认值已经是最佳配置了，所以最好不要改动太多配置，甚至可以忘记某些配置的存在。这就是为什么我们为本书创建了一个完整的最小的示例配置文件，可以作为自己的服务器配置文件的一个好的起点。有一些配置项是必选的，我们将在本章稍后解释。下面就是这个基础配置文件： 343

```
[mysqld]
# GENERAL
datadir                        = /var/lib/mysql
socket                         = /var/lib/mysql/mysql.sock
pid_file                       = /var/lib/mysql/mysql.pid
user                           = mysql
port                           = 3306
default_storage_engine         = InnoDB
# INNODB
innodb_buffer_pool_size        = <value>
innodb_log_file_size           = <value>
innodb_file_per_table          = 1
innodb_flush_method            = O_DIRECT
# MyISAM
key_buffer_size                = <value>
# LOGGING
log_error                      = /var/lib/mysql/mysql-error.log
slow_query_log                 = /var/lib/mysql/mysql-slow.log
# OTHER
tmp_table_size                 = 32M
max_heap_table_size            = 32M
query_cache_type               = 0
query_cache_size               = 0
max_connections                = <value>
thread_cache                   = <value>
table_cache                    = <value>
open_files_limit               = 65535
[client]
socket                         = /var/lib/mysql/mysql.sock
port                           = 3306
```

和你见过的其他配置文件[注6]相比，这里的配置选项可能太少了。但实际上已经超过了许多人的需要。有一些其他类型的配置选项可能也会用到，比如二进制日志，我们会在本章后面以及其他章节覆盖这些内容。

配置文件的第一件事是设置数据的位置。我们选择了 */var/lib/mysql* 路径存储数据，因为在许多类 UNIX 系统中这是最常见的位置。选择另外的位置也没有错，可以根据需要决定。我们把 PID 文件也放到相同的位置，但许多操作系统希望放在 */var/run* 目录下，这也可以。只需要简单地为这些选项配置一下就可以了。顺便说一下，不要把 Socket 文件和 PID 文件放到 MySQL 编译默认的位置，在不同的 MySQL 版本里这可能会导致一些错误。最好明确地设置这些文件的位置。（这么说并不是建议选择不同的位置，只是建
344 议确保在 *my.cnf* 文件中明确指定了这些文件的存放地点，这样升级 MySQL 版本时这些路径就不会改变。）

这里还指定了操作系统必须用 *mysql* 用户来运行 *mysqld* 进程。需要确保这个账户存在，并且拥有操作数据目录的权限。端口设置为默认的 3306，但有时可能需要修改一下。

注 6： 问：为排序缓存（Sort Buffer）和读缓存（Read Buffer）设置大小的选项在哪？答：它们已经很专注自己的事情了，除非觉得默认值不够好，否则保留默认值就可以了。

我们选择 InnoDB 作为默认的存储引擎，这个值得向大家解释一下。InnoDB 在大多数情况下是最好的选择，但并不总是如此。例如，一些第三方的软件，可能假设默认存储引擎是 MyISAM，所以创建表时没有指定存储引擎。这可能会导致软件故障，例如，这些应用可能会假定可以创建全文索引[注7]。默认存储引擎也会在显式创建临时表时用到，可能会引起服务器做一些意料之外的工作。如果希望持久化的表使用 InnoDB，但所有临时表使用 MyISAM，那应该确保在 CREATE TABLE 语句中明确指定了存储引擎。

一般情况下，如果决定使用一个存储引擎作为默认引擎，最好显式地进行配置。许多用户认为只使用了某个特定的存储引擎，但后来发现正在用的其实是另一个引擎，就是因为默认配置的是另外一个引擎。

接下来我们将阐述 InnoDB 的基础配置。InnoDB 在大多数情况下如果要运行得很好，配置大小合适的缓冲池（Buffer Pool）和日志文件（Log File）是必须的。默认值都太小了。其他所有的 InnoDB 设置都是可选的，尽管示例配置中因为可管理性和灵活性的原因启用了 innodb_file_per_table。设置 InnoDB 日志文件的大小和 innodb_flush_method 是本章后面要讨论的主题，其中 innodb_flush_method 是类 UNIX 系统特有的选项。

有一个流行的经验法则说，应该把缓冲池大小设置为服务器内存的约 75%~80%。这是另一个偶然有效的“比率”，但并不总是正确的。有一个更好的办法来设置缓冲池大小，大致如下：

1. 从服务器内存总量开始。
2. 减去操作系统的内存占用，如果 MySQL 不是唯一运行在这个服务器上的程序，还要扣掉其他程序可能占用的内存。
3. 减去一些 MySQL 自身需要的内存，例如为每个查询操作分配的一些缓冲。
4. 减去足够让操作系统缓存 InnoDB 日志文件的内存，至少是足够缓存最近经常访问的部分。（此建议适用于标准的 MySQL，Percona Server 可以配置日志文件用 O_DIRECT 方式打开，绕过操作系统缓存），留一些内存至少可以缓存二进制日志的最后一部分也是个很好的选择，尤其是如果复制产生了延迟，备库就可能读取主库上旧的二进制日志文件，给主库的内存造成一些压力。
5. 减去其他配置的 MySQL 缓冲和缓存需要的内存，例如 MyISAM 的键缓存（Key Cache），或者查询缓存（Query Cache）。 345
6. 除以 105%，这差不多接近 InnoDB 管理缓冲池增加的自身管理开销。
7. 把结果四舍五入，向下取一个合理的数值。向下舍入不会太影响结果，但是如果分配太多可能就会是件很糟糕的事情。

注 7： InnoDB 在 5.1/5.5 中都不支持全文索引，直到 5.6 版本 InnoDB 才支持全文索引。——译者注

我们对这里有些内存总量相关的问题有一点感到厌倦——什么是“操作系统的一个位(Bit)”？那是变化的，在本章和这本书的其余部分，我们将对此做一定深度的讨论。你必须了解你的系统，并且估算它需要多少内存才能良好地运转。这是为什么一个适合所有场景的配置文件是不存在的。经验，以及有时一点数学知识将给你提供指导。

下面是一个例子，假设有一个 192GB 内存的服务器，只运行 MySQL 并且只使用 InnoDB，没有查询缓存（Query Cache），也没有非常多的连接连到服务器。如果日志文件总大小是 4 GB，可能会像这样处理：“我认为所有内存的 5% 或者 2GB，取较大的那个，应该足够操作系统和 MySQL 的其他内存需求，为日志文件减去 4 GB，剩下的都给 InnoDB 用”。结果差不多是 177 GB，但是配置得稍微低一点可能是个好主意。比如可以先配置缓存池为 168GB。在服务器实际运行中若发现还有不少内存没有分配使用，在出于某些目的有机会重启时，可以再适当调大缓冲池的大小。

如果有大量 MyISAM 表需要缓存它们的索引，结果自然会有很大不同。在 Windows 下这也是完全不同的，大多数的 MySQL 版本在 Windows 下使用大内存都有问题（虽然在 MySQL 5.5 中有所改进），或者是出于某种原因不使用 O_DIRECT 也会有不同的结果。

正如你所看到的，从一开始就获得精确的设置并不是关键。从一个比默认值大一点但不是大得很离谱的安全值开始是比较好的，在服务器运行一段时间后，可以看看服务器真实情况需要使用多少内存。这些东西是很难预测，因为 MySQL 的内存利用率并不总是可以预测的：它可能依赖很多的因素，例如查询的复杂性和并发性。如果是简单的工作负载，MySQL 的内存需求是非常小的——大约 256 KB 的每个连接。但是，使用临时表、排序、存储过程等的复杂查询，可能使用更多的内存。

这就是我们选择一个非常安全的起点的原因。可以看到，即使是保守的 InnoDB 的缓冲池设置，实际上也是服务器内存的 87.5%——超过 75%，这就是为什么我们说简单地按比例是不正确的方法的原因。

我们建议，当配置内存缓冲区的时候，宁可谨慎，而不是把它们配置得过大。如果把缓冲池配置得比它可以设的值少了 20%，很可能只会对性能产生小的影响，也许就只影响
346 几个百分点。如果设置得大了 20%，则可能会造成更严重的问题：内存交换、磁盘抖动，甚至内存耗尽和硬件死机。

这份 InnoDB 配置的例子说明了我们配置服务器的首选途径：了解它内部做了什么，以及参数之间如何相互影响，然后再决定。

时间改变一切

精确地配置 MySQL 的内存缓冲区随着时间的推移变得不那么重要。当一个强大的服务器只有 4 GB 内存的时候，我们努力地平衡其资源使它可以运行 1 000 个连接。这通常需要我们为 MySQL 保留 1GB 的内存，这是服务器总内存的四分之一，而且会极大地影响我们设置缓冲池的大小。

如今类似的服务器有 144 GB 的内存，但是在大多数应用中我们通常看到的连接数是相同的，每个连接的缓冲区并没有真的改变太多。因此，我们可能会慷慨地为 MySQL 保留 4GB 的内存，这只是九牛一毛而已。它不会对我们的缓冲池的大小设置产生太大影响。

示例配置文件中的其他一些设置，大多是不言自明的，其中很多配置都是是与否的判断。在本章的其余部分，我们将探讨其中的几个。可以看到，我们已经启用日志记录、禁用了查询缓存，等等。在这一章的后面，我们还将讨论一些安全性和完整性的设置，它可以使服务器更强健，并对防止数据损坏和其他问题非常有帮助。我们并没有在这里展示这些设置。

这里需要解释的一个选项是 `open_files_limit`。在典型的 Linux 系统上我们把它设置得尽可能大。现代操作系统中打开文件句柄开销都很小。如果这个参数不够大，将会碰到经典的 24 号错误，“打开的文件太多（too many open files）”。

跳过其他的直接看到末尾，在配置文件的最后一节，是为了如 *mysql* 和 *mysqladmin* 之类的客户端程序做的设置，可以简化这些程序连接到服务器的步骤。应该为客户端程序设置那些匹配服务器的配置项。

8.3.1 检查 MySQL 服务器状态变量

有时可以使用 `SHOW GLOBAL STATUS` 的输出，作为配置的输入，以更好地通过工作负载来自定义配置。为了达到最佳效果，既要看绝对值，又要看值是如何随时间而改变的，最好为高峰和非高峰时间的值做几个快照。可以使用以下命令每隔 60 秒来查看状态变量的增量变化：

```
$ mysqladmin extended-status -ri60
```

在解释配置设置的时候，我们经常会提到随着时间的推移各种状态变量的变化。所以通常可以预料到需要分析如刚才那个命令的输出的情况。有一些有用的工具，如 Percona 347

Toolkit 中的 *pt-mext* 或 *PT-mysql-summary*，可以简洁地显示状态计数器的变化，不用直接看那些 SHOW 命令的输出。

好吧，前面的内容算是预热，接下来我们将进入一些服务器内核的东西，并将相关的配置建议穿插在其中。然后再回头来看示例配置文件，就会有足够的背景知识来选择适当的配置选项的值了。

8.4 配置内存使用

配置 MySQL 正确地使用内存量对高性能是至关重要的。肯定要根据需求来定制内存使用。可以认为 MySQL 的内存消耗分为两类：可以控制的内存和不可以控制的内存。无法控制 MySQL 服务器运行、解析查询，以及其内部管理所消耗的内存，但是为特定目的而使用多少内存则有很多参数可以控制[注8]。用好可以控制的内存并不难，但需要对配置的含义非常清楚。

像前面展示的，按下面的步骤来配置内存：

1. 确定可以使用的内存上限。
2. 确定每个连接 MySQL 需要使用多少内存，例如排序缓冲和临时表。
3. 确定操作系统需要多少内存才够用。包括同一台机器上其他程序使用的内存，如定时任务。
4. 把剩下的内存全部给 MySQL 的缓存，例如 InnoDB 的缓冲池，这样做很有意义。

我们将在后面的章节详细说明这些步骤，然后我们对各种 MySQL 的缓存需求做更细节的分析。

8.4.1 MySQL 可以使用多少内存

在任何给定的操作系统上，MySQL 都有允许使用的内存上限。基本出发点是机器上安装了多少物理内存。如果服务器就没装这么多内存，MySQL 肯定也不能用这么多内存。

还需要考虑操作系统或架构的限制，如 32 位操作系统对一个给定的进程可以处理多少内存是有限制的。因为 MySQL 是单进程多线程的运行模式，它整体可用的内存量也许会
348 受操作系统位数的严格限制——例如，32 位 Linux 内核通常限制任意进程可以使用的内存量在 2.5GB ~ 2.7GB 范围内。运行时地址空间溢出是非常危险的，可能导致 MySQL 崩溃。现在这种情况非常难得一见，但以前这种情况很常见。

有许多其他的操作系统——特殊的参数和古怪的事情必须考虑到，例如不只是每个进

注 8： 例如 Join Buffer / Sort Buffer 等。——译者注

程有限制，而且堆栈大小和其他设置也有限制。系统的 *glibc* 库也可能限制每次分配的内存大小。例如，若 *glibc* 库支持单次分配的最大大小是 2GB，那么可能就无法设置 `innodb_buffer_pool` 的值大于 2 GB。

即使在 64 位服务器上，依然有一些限制。例如，许多我们讨论的缓冲区，如键缓存（Key Buffer），在 5.0 以及更早的 MySQL 版本上，有 4GB 的限制，即使在 64 位服务器上也是如此。在 MySQL 5.1 中，部分限制被取消了，在 MySQL 手册中记载了每个变量的最大值，有需要可以查阅。

8.4.2 每个连接需要的内存

MySQL 保持一个连接（线程）只需要少量的内存。它还要求一个基本量的内存来执行任何给定查询。你需要为高峰时期执行的大量查询预留好足够的内存。否则，查询执行可能因为缺乏内存而导致执行效率不佳或执行失败。

知道在高峰时期 MySQL 将消耗多少内存是非常有用的，但一些习惯性用法可能意外地消耗大量内存，这使得对内存使用量的预测变得比较困难。绑定变量就是一个例子，因为可以一次打开很多绑定变量语句。另一个例子是 InnoDB 数据字典（关于这个后面我们再细说）。

当预测内存峰值消耗时，没必要假设一个最坏情况。例如，配置 MySQL 允许最多 100 个连接，在理论上可能出现 100 个连接同时在运行很大的查询，但在现实情况中，这可能不会发生。例如，设置 `myisam_sort_buffer_size` 为 256MB，最差情况下至少需要使用 25 GB 内存，但这种最差情况在实际中几乎是不可能发生的。使用了许多大的临时表或复杂存储过程的查询，通常是导致高内存消耗最可能的原因。

相对于计算最坏情况下的开销，更好的办法是观察服务器在真实的工作压力下使用了多少内存，可以在进程的虚拟内存大小那里看到。在许多类 UNIX 系统里，可以观察 *top* 命令中的 `VIRT` 列，或者 *ps* 命令中的 `VSZ` 列的值。下一章有更多关于如何监视内存使用情况的信息。

8.4.3 为操作系统保留内存 349

跟查询一样，操作系统也需要保留足够的内存给它工作。如果没有虚拟内存正在交换（Paging）到磁盘，就是表明操作系统内存足够的最佳迹象。（关于这个话题，请参阅下一章。）

至少应该为操作系统保留 1GB ~ 2GB 的内存——如果机器内存更多就再多预留一些。我们建议 2 GB 或总内存的 5% 作为基准，以较大者为准。为了安全再额外增加一些预留，并且如果机器上还在运行内存密集型任务（如备份），则可以再多增加一些预留。不要为操作系统的缓存增加任何内存，因为它们可能会变得非常大。操作系统通常会利用所有剩下的内存来做文件系统缓存，我们认为，这应该从操作系统自身的需求里分离出来。

8.4.4 为缓存分配内存

如果服务器只运行 MySQL，所有不需要为操作系统以及查询处理保留的内存都可以用作 MySQL 缓存。

相比其他，MySQL 需要为缓存分配更多的内存。它使用缓存来避免磁盘访问，磁盘访问比内存访问数据要慢得多。操作系统可能会缓存一些数据，这对 MySQL 有些好处（尤其是对 MyISAM），但是 MySQL 自身也需要大量内存。

下面是我们认为对大部分情况来说最重要的缓存：

- InnoDB 缓冲池
- InnoDB 日志文件和 MyISAM 数据的操作系统缓存
- MyISAM 键缓存
- 查询缓存
- 无法手工配置的缓存，例如二进制日志和表定义文件的操作系统缓存

还有些其他缓存，但是它们通常不会使用太多内存。我们在前面的章节中讨论了查询缓存（Query Cache）的细节，所以接下来的部分我们专注于 InnoDB 和 MyISAM 良好工作需要的缓存。

如果只使用单一存储引擎，配置服务器就简单多了。如果只使用 MyISAM 表，就可以完全关闭 InnoDB，而如果只使用 InnoDB，就只需要分配最少的资源给 MyISAM（MySQL 内部系统表采用 MyISAM）。但是如果正混合使用各种存储引擎，就很难在它们之间找到恰当的平衡。我们发现最好的办法是先做一个有根据的猜测，然后在运行中观察服务器（再进行调整）。

350

8.4.5 InnoDB 缓冲池（Buffer Pool）

如果大部分都是 InnoDB 表，InnoDB 缓冲池或许比其他任何东西更需要内存。InnoDB 缓冲池并不仅仅缓存索引：它还会缓存行的数据、自适应哈希索引、插入缓冲（Insert Buffer）、锁，以及其他内部数据结构。InnoDB 还使用缓冲池来帮助延迟写入，这样就能合并多个写入操作，然后一起顺序地写回。总之，InnoDB 严重依赖缓冲池，你必须

确认为它分配了足够的内存，通常就像这一章前面展示的那样处理。可以使用通过 SHOW 命令得到的变量或者例如 *innotop* 这样的工具监控 InnoDB 缓冲池的内存利用情况。

如果数据量不大，并且不会快速增长，就没必要为缓冲池分配过多的内存。把缓冲池配置得比需要缓存的表和索引还要大很多实际上没有什么意义。当然，对一个迅速增长的数据库做超前的规划没有错，但有时我们也会看到一个巨大的缓冲池只缓存一点点数据，这就没有必要了。

很大的缓冲池也会带来一些挑战，例如，预热和关闭都会花费很长的时间。如果有很多脏页在缓冲池里，InnoDB 关闭时可能会花费较长的时间，因为在关闭之前需要把脏页写回数据文件。也可以强制快速关闭，但是重启时就必须多做更多的恢复工作，也就是说无法同时加速关闭和重启两个动作。如果事先知道什么时候需要关闭 InnoDB，可以在运行时修改 `innodb_max_dirty_pages_pct` 变量，将值改小，等待刷新线程清理缓冲池，然后在脏页数量较少时关闭。可以监控 `the Innodb_buffer_pool_pages_dirty` 状态变量或者使用 *innotop* 来监控 `SHOW INNODB STATUS` 来观察脏页的刷新量。

更小的 `innodb_max_dirty_pages_pct` 变量值并不保证 InnoDB 将在缓冲池中保持更少的脏页。它只是控制 InnoDB 是否可以"偷懒（Lazy）"的阈值。InnoDB 默认通过一个后台线程来刷新脏页，并且会合并写入，更高效地顺序写出到磁盘。这个行为之所以被称为"偷懒（Lazy）"，是因为它使得 InnoDB 延迟了缓冲池中刷写脏页的操作，直到一些其他数据必须使用空间时才刷写。当脏页的百分比超过了这个阈值，InnoDB 将快速地刷写脏页，尝试让脏页的数量更低。当事务日志没有足够的空间剩余时，InnoDB 也将进入"激烈刷写（Furious Flushing）"模式，这就是大日志可以提升性能的一个原因。

当有一个很大的缓冲池，重启后服务器也许需要花很长的时间（几个小时甚至几天）来预热缓冲池，尤其是磁盘很慢的时候。在这种情况下，可以利用 Percona Server 的功能来重新载入缓冲池的页[注9]，从而节省时间。这可以让预热时间减少到几分钟。MySQL 5.6 也提供了一个类似的功能。这个功能对复制尤其有好处，因为单线程复制导致备库需要额外的预热时间。

如果不能使用 Percona Server 的快速预热功能，也可以在重启后立刻进行全表扫描或者 351
索引扫描，把索引载入缓冲池。这是比较粗暴的方式，但是有时候比什么都不做还是要好。可以使用 `init_file` 设置来实现这个功能。把 SQL 放到一个文件里，然后当 MySQL 启动的时候来执行。文件名必须在 `init_file` 选项中指定，文件中可以包含多条 SQL 命令，每一条单独一行（不允许使用注释）。

注 9： 这个功能是 Dump/Restore of the Buffer Pool，详情查看：*http://www.percona.com/doc/percona-server/5.5/management/innodb_lru_dump_restore.html*。——译者注

8.4.6 MyISAM 键缓存（Key Caches）

MyISAM 的键缓存也被称为键缓冲，默认只有一个键缓存，但也可以创建多个。不像 InnoDB 和其他一些存储引擎，MyISAM 自身只缓存索引，不缓存数据（依赖操作系统缓存数据）。如果大部分是 MyISAM 表，就应该为键缓存分配比较多的内存。

最重要的配置项是 key_buffer_size。任何没有分配给它的内存[注10]都可以被操作系统缓存利用。MySQL 5.0 有一个规定的有效上限是 4GB，不管系统是什么架构。MySQL 5.1 允许更大的值。可以查看正在使用的 MySQL 版本的官方手册来了解这个限制。

在决定键缓存需要分配多少内存之前，先去了解 MyISAM 索引实际上占用多少磁盘空间是很有帮助的。肯定不需要把键缓冲设置得比需要缓存的索引数据还大。查询 INFORMATION_SCHEMA 表的 INDEX_LENGTH 字段，把它们的值相加，就可以得到索引存储占用的空间：

```
SELECT SUM(INDEX_LENGTH) FROM INFORMATION_SCHEMA.TABLES WHERE ENGINE='MYISAM';
```

如果是类 UNIX 系统，也可以使用下面的命令：

```
$ du -sch `find /path/to/mysql/data/directory/ -name "*.MYI"`
```

应该把键缓存设置得多大？不要超过索引的总大小，或者不超过为操作系统缓存保留总内存的 25% ～ 50%，以更小的为准。

默认情况下，MyISAM 将所有索引都缓存在默认键缓存中，但也可以创建多个命名的键缓冲。这样就可以同时缓存超过 4GB 的内存。如果要创建名为 key_buffer_1 和 key_buffer_2 的键缓冲，每个大小为 1GB，则可以在配置文件中添加如下配置项：

```
key_buffer_1.key_buffer_size = 1G
key_buffer_2.key_buffer_size = 1G
```

现在有了三个键缓冲：两个由这两行配置明确定义，还有一个是默认键缓冲。可以使用 CACHE INDEX 命令来将表映射到对应的缓冲区。使用下面的语句，让 MySQL 使用 key_buffer_1 缓冲区来缓存 t1 和 t2 表的索引：

```
mysql> CACHE INDEX t1, t2 IN key_buffer_1;
```

352 现在当 MySQL 从这些表的索引读取块时，将会在指定的缓冲区内缓存这些块。也可以把表的索引预载入到缓存中，通过 init_file 设置或者 LOAD INDEX 命令：

```
mysql> LOAD INDEX INTO CACHE t1, t2;
```

注 10：当然还要排除各种操作系统自身占用的内存，还有 MySQL 自身占用的内存等。——译者注

任何没明确指定映射到哪个键缓冲区的索引，在 MySQL 第一次需要访问 *.MYI* 文件的时候，都会被分配到默认缓冲区。

可以通过 SHOW STATUS 和 SHOW VARIABLES 命令的信息来监控键缓冲的使用情况。下面的公式可以计算缓冲区的使用率：

```
100 - ( (Key_blocks_unused * key_cache_block_size) * 100 / key_buffer_size )
```

如果服务器运行了很长一段时间后，还是没有使用完所有的键缓冲，就可以把缓冲区调小一点。

键缓冲命中率有什么意义？正如我们之前解释的那样，这个数字没什么用。例如，99% 和 99.9% 之间看起来差别很小，但实际上代表了 10 倍的差距。缓存命中率也是和应用相关的：有些应用可以在 95% 的命中率下工作良好，但是也有些应用可能是 I/O 密集型的，必须在 99.9% 的命中率下工作。甚至有可能在恰当大小的缓存设置下获得 99.99% 的命中率。

从经验上来说，每秒缓存未命中的次数要更有用。假定有一个独立的磁盘，每秒可以做 100 个随机读。每秒 5 次缓存未命中可能不会导致 I/O 繁忙，但是每秒 80 次缓存未命中则可能出现问题。可以使用下面的公式来计算这个值：

```
Key_reads / Uptime
```

通过间隔 10 ~ 100 秒来计算这段时间内缓存未命中次数的增量值，可以获得当前性能的情况。下面的命令可以每 10 秒钟获取一次状态值的变化量：

```
$ mysqladmin extended-status -r -i 10 | grep Key_reads
```

记住，MyISAM 使用操作系统缓存来缓存数据文件，通常数据文件比索引要大。因此，把更多的内存保留给操作系统缓存而不是键缓存是有意义的。即使你有足够的内存来缓存所有索引，并且键缓存命中率很低，当 MySQL 尝试读取数据文件时（不是索引文件），在操作系统层还是可能发生缓存未命中，这对 MySQL 完全透明，MySQL 并不能感知到。因此，这种情况下可能会有大量数据文件缓存未命中，这和索引的键缓存未命中率是完全不相关的。

最后，即使没有任何 MyISAM 表，依然需要将 key_buffer_size 设置为较小的值，例如 32M。MySQL 服务器有时会在内部使用 MyISAM 表，例如 GROUP BY 语句可能会使用 MyISAM 做临时表。

353 **MySQL 键缓存块大小（Key Block Size）**

块大小也是很重要的（尤其是写密集型负载），因为它影响了 MyISAM、操作系统缓存，以及文件系统之间的交互。如果缓存块太小了，可能会碰到写时读取（read-around write），就是操作系统在执行写操作之前必须先从磁盘上读取一些数据。下面说明一下这种情况是怎么发生的，假设操作系统的页大小是 4KB（在 x86 架构上通常都是这样），并且索引块大小是 1KB：

1. MyISAM 请求从磁盘上读取 1KB 的块。
2. 操作系统从磁盘上读取 4KB 的数据并缓存，然后发送需要的 1KB 数据给 MyISAM。
3. 操作系统丢弃缓存数据以给其他数据腾出缓存。
4. MyISAM 修改 1KB 的索引块，然后请求操作系统把它写回磁盘。
5. 操作系统从磁盘读取同一个 4KB 的数据，写入操作系统缓存，修改 MyISAM 改动的这 1KB 数据，然后把整个 4KB 的块写回磁盘。

在第 5 步中，当 MyISAM 请求操作系统去写 4KB 页的部分内容时，就发生了写时读取（read-around write）。如果 MyISAM 的块大小跟操作系统的相匹配，在第 5 步的磁盘读就可以避免[注11]。

很遗憾，MySQL 5.0 以及更早的版本没有办法配置索引块大小。但是，在 MySQL 5.1 以及更新版本中，可以设置 MyISAM 的索引块大小跟操作系统一样，以避免写时读取。`myisam_block_size` 变量控制着索引块大小。也可以指定每个索引的块大小，在 `CREATE TABLE` 或者 `CREATE INDEX` 语句中使用 `KEY_BLOCK_SIZE` 选项即可，但是因为同一个表的所有索引都保存在同一个文件中，因此该表所有索引的块大小都需要大于或者等于操作系统的块大小，才能避免由于边界对齐导致的写时读取。（例如，若同一个表的两个索引，一个块大小是 1KB，另一个是 4KB。那么 4KB 的索引块边界很可能和操作系统的页边界是不对齐的，这样还是会发生写时读取。）

8.4.7 线程缓存

线程缓存保存那些当前没有与连接关联但是准备为后面新的连接服务的线程。当一个新的连接创建时，如果缓存中有线程存在，MySQL 从缓存中删除一个线程，并且把它分配给这个新的连接。当连接关闭时，如果线程缓存还有空间的话，MySQL 又会把线程
354 放回缓存。如果没有空间的话，MySQL 会销毁这个线程。只要 MySQL 在缓存里还有空闲的线程，它就可以迅速地响应连接请求，因为这样就不用为每个连接创建新的线程。

注 11： 理论上，如果能确认原生 4KB 的数据依然在操作系统缓存中，读操作就不需要了。然而，你没法控制操作系统把哪些块放到缓存中。通过 *fincore* 工具可以看到哪些块在缓存中，地址在：*http://net.doit.wisc.edu/~plonka/fincore/*。

thread_cache_size 变量指定了 MySQL 可以保持在缓存中的线程数。一般不需要配置这个值，除非服务器会有很多连接请求。要检查线程缓存是否足够大，可以查看 Threads_created 状态变量。如果我们观察到很少有每秒创建的新线程数少于 10 个的时候，通常应该尝试保持线程缓存足够大，但是实际上经常也可能看到每秒少于 1 个新线程的情况。

一个好的办法是观察 Threads_connected 变量并且尝试设置 thread_cache_size 足够大以便能处理业务压力正常的波动。例如，若 Threads_connected 通常保持在 100 ~ 120，则可以设置缓存大小为 20。如果它保持在 500 ~ 700，200 的线程缓存应该足够大了。可以这样认为：在 700 个连接的时候，可能没有线程在缓存中；在 500 个连接的时候，有 200 个缓存的线程准备为负载再次增加到 700 个连接时使用。

把线程缓存设置得非常大在大部分时候是没有必要的，但是设置得很小也不能节省太多内存，所以也没什么好处。每个在线程缓存中的线程或者休眠状态的线程，通常使用 256KB 左右的内存。相对于正在处理查询的线程来说，这个内存不算很大。通常应该保证线程缓存足够大，以避免 Threads_created 频繁增长。如果这个数字很大（例如，几千个线程），可能需要把 thread_cache_size 设置得稍微小一些，因为一些操作系统不能很好地处理庞大的线程数，即使其中大部分是休眠的。

8.4.8 表缓存（Table Cache）

表缓存和线程缓存的概念是相似的，但存储的对象代表的是表。每个在缓存中的对象包含相关表 *.frm* 文件的解析结果，加上一些其他数据。准确地说，在对象里的其他数据的内容依赖于表的存储引擎。例如，对 MyISAM，是表的数据和索引的文件描述符。对于 Merge 表则可能是多个文件描述符，因为 Merge 表可以有很多的底层表。

表缓存可以重用资源。举个实际的例子，当一个查询请求访问一张 MyISAM 表，MySQL 也许可以从缓存的对象中获取到文件描述符。尽管这样做可以避免打开一个文件描述符的开销，但这个开销其实并不大。打开和关闭文件描述符在本地存储是很快的，服务器可以轻松地完成每秒 100 万次的操作（尽管这跟网络存储不同）。对 MyISAM 表来说，表缓存的真正好处是，可以让服务器避免修改 MyISAM 文件头来标记表“正在使用中”[注12]。

表缓存的设计是服务器和存储引擎之间分离不彻底的产物，属于历史问题。表缓存对 InnoDB 重要性就小多了，因为 InnoDB 不依赖它来做那么多的事（例如持有文件描述符， 355

注 12：“打开的表（Opened Table）”的概念，可能有点混乱。当不同的查询同时访问一张表，或者是一个单独的查询引用同一张表超过一次，比如子查询或者自关联，MySQL 都会对一张表作为打开状态多次计数。MyISAM 表的索引文件包含一个计数器，MyISAM 表打开时递增，关闭时递减。这使得对于 MyISAM 表可以看到是不是关闭干净了：如果首次打开一个表，计数器不为零，说明表没有关闭干净。

InnoDB 有自己的表缓存版本）。尽管如此，InnoDB 也能从缓存解析的 *.frm* 文件中获益。

在 MySQL 5.1 版本中，表缓存分离成两部分：一个是打开表的缓存，一个是表定义缓存（通过 table_open_cache 和 table_definition_cache 变量来配置）。其结果是，表定义（解析 .frm 文件的结果）从其他资源中分离出来了，例如表描述符。打开的表依然是每个线程、每个表用的，但是表定义是全局的，可以被所有连接有效地共享。通常可以把 table_definition_cache 设置得足够高，以缓存所有的表定义。除非有上万张表，否则这可能是最简单的方法。

如果 Opened_tables 状态变量很大或者在增长，可能是因为表缓存不够大，那么可以人为增加 table_cache 系统变量（或者是 MySQL 5.1 中的 table_open_cache）。然而，当创建和删除临时表时，要注意这个计数器的增长，如果经常需要创建和删除临时表，那么该计数器就会不停地增长。

把表缓存设置得非常大的缺点是，当服务器有很多 MyISAM 表时，可能会导致关机时间较长，因为关机前索引块必须完成刷新，表都必须标记为不再打开。同样的原因，也可能使 FLUSH TABLES WITH READ LOCK 操作花费很长一段时间。更为严重的是，检查表缓存算法不是很有效，稍后会更详细地说明。

如果遇到 MySQL 无法打开更多文件的错误（可以使用 *perror* 工具来检查错误号代表的含义），那么可能需要增加 MySQL 允许打开文件的数量。这可以通过在 *my.cnf* 文件中设置 open_files_limit 服务器变量来实现。

线程和表缓存实际上用的内存并不多，相反却可以有效节约资源。虽然创建一个新线程或者打开一个新的表，相对于其他 MySQL 操作来说代价并不算高，但它们的开销是会累加的。所以缓存线程和表有时可以提升效率。

356

8.4.9 InnoDB 数据字典（Data Dictionary）

InnoDB 有自己的表缓存，可以称为表定义缓存或者数据字典，在目前的 MySQL 版本中还不能对它进行配置。当 InnoDB 打开一张表，就增加了一个对应的对象到数据字典。每张表可能占用 4KB 或者更多的内存（尽管在 MySQL 5.1 中对空间的需求小了很多）。当表关闭的时候也不会从数据字典中移除它们。

因此，随着时间的推移，服务器可能出现内存泄露，导致数据字典中的元素不断地增长。但这不是真的内存泄露，只是没有对数据字典实现任何一种缓存过期策略。通常只有当有很多（数千或数万）张大表时才是个问题。如果这个问题有影响，可以使用 Percona Server，有一个选项可以控制数据字典的大小，它会从数据字典中移除没有使用的表。MySQL 5.6 尚未发布的版本中也有个类似的功能。

另一个性能问题是第一次打开表时会计算统计信息，这需要很多 I/O 操作，所以代价很高。相比 MyISAM，InnoDB 没有将统计信息持久化，而是在每次打开表时重新计算，在打开之后，每隔一段过期时间或者遇到触发事件（改变表的内容或者查询 INFORMATION_SCHEMA 表，等等），也会重新计算统计信息。如果有很多表，服务器可能会花费数个小时来启动并完全预热，在这个时候服务器可能花费更多的时间在等待 I/O 操作，而不是做其他事。可以在 Percona Server（在 MySQL 5.6 中也可以，但是叫做 innodb_analyze_is_persistent）中打开 innodb_use_sys_stats_table 选项来持久化存储统计信息到磁盘，以解决这个问题。

即使在启动之后，InnoDB 统计操作还可能对服务器和一些特定的查询产生冲击。可以关闭 innodb_stats_on_metadata 选项来避免耗时的表统计信息刷新。当例如 IDE 这样的工具执行 INFORMATION_SCHEMA 表的查询时，关闭这个选项后的表现是很不一样的（当然是快了不少）。

如果设置了 InnoDB 的 innodb_file_per_table 选项（后面会描述），InnoDB 任意时刻可以保持打开 *.ibd* 文件的数量也是有其限制的。这由 InnoDB 存储引擎负责，而不是 MySQL 服务器管理，并且由 innodb_open_files 来控制。InnoDB 打开文件和 MyISAM 的方式不一样，MyISAM 用表缓存来持有打开表的文件描述符，而 InnoDB 在打开表和打开文件之间没有直接的关系。InnoDB 为每个 *.ibd* 文件使用单个、全局的文件描述符。如果可以，最好把 innodb_open_files 的值设置得足够大以使服务器可以保持所有的 *.ibd* 文件同时打开。

8.5 配置 MySQL 的 I/O 行为

有一些配置项影响着 MySQL 怎样同步数据到磁盘以及如何做恢复操作。这些操作对性能的影响非常大，因为都涉及到昂贵的 I/O 操作。它们也表现了性能和数据安全之间的权衡。通常，保证数据立刻并且一致地写到磁盘是很昂贵的。如果能够冒一点磁盘写可 357
能没有真正持久化到磁盘的风险，就可以增加并发性和减少 I/O 等待，但是必须决定可以容忍多大的风险。

8.5.1 InnoDB I/O 配置

InnoDB 不仅允许控制怎么恢复，还允许控制怎么打开和刷新数据（文件），这会对恢复和整体性能产生巨大的影响。尽管可以影响它的行为，InnoDB 的恢复流程实际上是自动的，并且经常在 InnoDB 启动时运行。撇开恢复并假设 InnoDB 没有崩溃或者出错，InnoDB 依然有很多需要配置的地方。它有一系列复杂的缓存和文件设计可以提升性能，以及保证 ACID 特性，并且每一部分都是可配置的，图 8-1 阐述了这些文件和缓存。

对于常见的应用，最重要的一小部分内容是 InnoDB 日志文件大小、InnoDB 怎样刷新它的日志缓冲，以及 InnoDB 怎样执行 I/O。

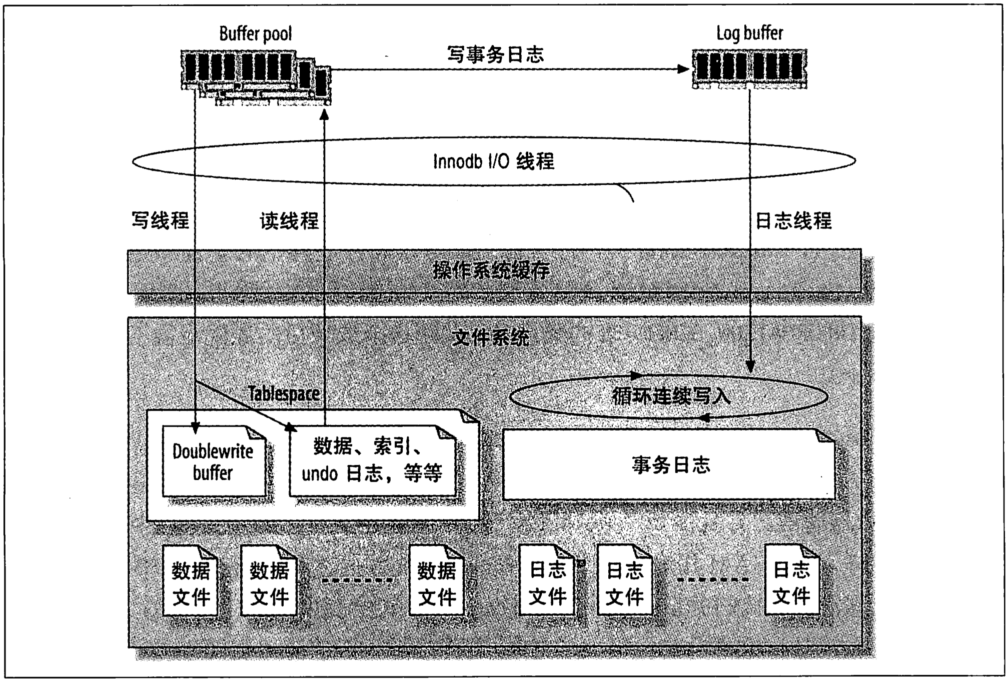


图8-1：InnoDB的缓存和文件

InnoDB 事务日志

InnoDB 使用日志来减少提交事务时的开销。因为日志中已经记录了事务，就无须在每个事务提交时把缓冲池的脏块刷新（flush）到磁盘中。事务修改的数据和索引通常会映射到表空间的随机位置，所以刷新这些变更到磁盘需要很多随机 I/O。InnoDB 假设使用的是常规磁盘（机械磁盘），随机 I/O 比顺序 I/O 要昂贵得多，因为一个 I/O 请求需要时间把磁头移到正确的位置，然后等待磁盘上读出需要的部分，再转到开始位置。

InnoDB 用日志把随机 I/O 变成顺序 I/O。一旦日志安全写到磁盘，事务就持久化了，即使变更还没写到数据文件。如果一些糟糕的事情发生了（例如断电了），InnoDB 可以重放日志并且恢复已经提交的事务。

当然，InnoDB 最后还是必须把变更写到数据文件，因为日志有固定的大小。InnoDB 的日志是环形方式写的：当写到日志的尾部，会重新跳转到开头继续写，但不会覆盖还没应用到数据文件的日志记录，因为这样做会清掉已提交事务的唯一持久化记录。

InnoDB使用一个后台线程智能地刷新这些变更到数据文件。这个线程可以批量组合写入，使得数据写入更顺序，以提高效率。实际上，事务日志把数据文件的随机I/O转换为几乎顺序的日志文件和数据文件I/O。把刷新操作转移到后台使查询可以更快完成，并且缓和查询高峰时I/O系统的压力。

整体的日志文件大小受控于innodb_log_file_size和innodb_log_files_in_group两个参数，这对写性能非常重要。日志文件的总大小是每个文件的大小之和。默认情况下，只有两个5MB的文件，总共10MB。对高性能工作来说这太小了。至少需要几百MB，或者甚至上GB的日志文件。 358

InnoDB使用多个文件作为一组循环日志。通常不需要修改默认的日志数量，只修改每个日志文件的大小即可。要修改日志文件大小，需要完全关闭MySQL，将旧的日志文件移到其他地方保存，重新配置参数，然后重启。一定要确保MySQL干净地关闭了，或者还有日志文件可以保证需要应用到数据文件的事务记录，否则数据库就无法恢复了！当重启服务器的时候，查看MySQL的错误日志。在重启成功之后，才可以删除旧的日志文件。

日志文件大小和日志缓存。要确定理想的日志文件大小，必须权衡正常数据变更的开销和崩溃恢复需要的时间。如果日志太小，InnoDB将必须做更多的检查点，导致更多的日志写。在极个别情况下，写语句可能被拖累，在日志没有空间继续写入前，必须等待变更被应用到数据文件。另一方面，如果日志太大了，在崩溃恢复时InnoDB可能不得不做大量的工作。这可能极大地增加恢复时间，尽管这个处理在新的MySQL版本中已经改善很多。

数据大小和访问模式也将影响恢复时间。假设有一个1TB的数据和16GB的缓冲池，并且全部日志大小是128MB。如果缓冲池里有很多脏页（例如，页被修改了还没被刷写回数据文件），并且它们均匀地分布在1TB数据中，崩溃后恢复将需要相当长一段时间。InnoDB必须从头到尾扫描日志，仔细检查数据文件，如果需要还要应用变更到数据文件。这是很庞大的读写操作！另一方面，如果变更是局部性的——就是说，如果只有几百MB数据被频繁地变更——恢复可能就很快，即使数据和日志文件很大。恢复时间也依赖于普通修改操作的大小，这跟数据行的平均长度有关系。较短的行使得更多的修改可以放在同样的日志中，所以InnoDB可能必须在恢复时重放更多修改操作[注13]。 359

当InnoDB变更任何数据时，会写一条变更记录到内存日志缓冲区。在缓冲满的时候、事务提交的时候，或者每一秒钟，InnoDB都会刷写缓冲区的内容到磁盘日志文件——无论上述三个条件哪个先达到。如果有大事务，增加日志缓冲区（默认1MB）大小可以

注13：对于好奇的人，Percona Server的innodb_recovery_stats选项可以帮助你从执行崩溃恢复的立场来理解服务器的工作负载。

帮助减少 I/O。变量 innodb_log_buffer_size 可以控制日志缓冲区的大小。

通常不需要把日志缓冲区设置得非常大。推荐的范围是 1MB ~ 8MB，一般来说足够了，除非要写很多相当大的 BLOB 记录。相对于 InnoDB 的普通数据，日志条目是非常紧凑的。它们不是基于页的，所以不会浪费空间来一次存储整个页。InnoDB 也使得日志条目尽可能地短。有时甚至会保存为函数号和 C 函数的参数！

较大的日志缓冲区在某些情况下也是有好处的：可以减少缓冲区中空间分配的争用。当配置一台有大内存的服务器时，有时简单地分配 32MB ~ 128MB 的日志缓冲，因为花费这么点相对（整机）而言比较小的内存并没有什么不好，还可以帮助避免压力瓶颈。如果有问题，瓶颈一般会表现为日志缓冲 Mutex 的竞争。

可以通过检查 SHOW INNODB STATUS 的输出中 LOG 部分来监控 InnoDB 的日志和日志缓冲区的 I/O 性能，通过观察 Innodb_os_log_written 状态变量来查看 InnoDB 对日志文件写出了多少数据。一个好用的经验法则是，查看 10 ~ 100 秒间隔的数字，然后记录峰值。可以用这个来判断日志缓冲是否设置得正好。例如，若看到峰值是每秒写 100KB 数据到日志，那么 1MB 的日志缓冲可能足够了。也可以使用这个衡量标准来决定日志文件设置多大会比较好。如果峰值是 100KB/s，那么 256MB 的日志文件足够存储至少 2 560 秒
360 的日志记录。这看起来足够了。作为一个经验法则，日志文件的全部大小，应该足够容纳服务器一个小时的活动内容。

InnoDB 怎样刷新日志缓冲。当 InnoDB 把日志缓冲刷新到磁盘日志文件时，先会使用一个 Mutex 锁住缓冲区，刷新到所需要的位置，然后移动剩下的条目到缓冲区的前面。当 Mutex 释放时，可能有超过一个事务已经准备好刷新其日志记录。InnoDB 有一个 Group Commit 功能，可以在一个 I/O 操作内提交多个事务，但是在 MySQL 5.0 中当打开二进制日志时这个功能就不能用了。我们在前一章写了一些关于 Group Commit 的东西。

日志缓冲必须被刷新到持久化存储，以确保提交的事务完全被持久化了。如果和持久相比更在乎性能，可以修改 innodb_flush_log_at_trx_commit 变量来控制日志缓冲刷新的频繁程度。可能的设置如下：

0

把日志缓冲写到日志文件，并且每秒钟刷新一次，但是事务提交时不做任何事。

1

将日志缓冲写到日志文件，并且每次事务提交都刷新到持久化存储。这是默认的（并且是最安全的）设置，该设置能保证不会丢失任何已经提交的事务，除非磁盘或者操作系统是“伪”刷新。

2
: 每次提交时把日志缓冲写到日志文件，但是并不刷新。InnoDB 每秒钟做一次刷新。0 与 2 最重要的不同是（也是为什么 2 是更合适的设置），如果 MySQL 进程“挂了”，2 不会丢失任何事务。如果整个服务器“挂了”或者断电了，则还是可能会丢失一些事务。

了解清楚“把日志缓冲写到日志文件”和“把日志刷新到持久化存储”之间的不同是很重要的。在大部分操作系统中，把缓冲写到日志只是简单地把数据从 InnoDB 的内存缓冲转移到了操作系统的缓存，也是在内存里，并没有真的把数据写到了持久化存储。

因此，如果 MySQL 崩溃了或者电源断电了，设置 0 和 2 通常会导致最多一秒的数据丢失，因为数据可能只存在于操作系统的缓存。我们说“通常”，因为不论如何 InnoDB 会每秒尝试刷新日志文件到磁盘，但是在一些场景下也可能丢失超过 1 秒的事务，例如当刷新被推迟了。

与此相反，把日志刷新到持久化存储意味着 InnoDB 请求操作系统把数据刷出缓存，并且确认写到磁盘了。这是一个阻塞 I/O 的调用，直到数据被完全写回才会完成。因为写数据到磁盘比较慢，当 `innodb_flush_log_at_trx_commit` 被设置为 1 时，可能明显地降 361
低 InnoDB 每秒可以提交的事务数。今天的高速驱动器[注 14]可能每秒只能执行一两百个磁盘事务，受限于磁盘旋转速度和寻道时间。

有时硬盘控制器或者操作系统假装做了刷新，其实只是把数据放到了另一个缓存，例如磁盘自己的缓存。这更快但是很危险，因为如果驱动器断电，数据依然可能丢失。这甚至比设置 `innodb_flush_log_at_trx_commit` 为不为 1 的值更糟糕，因为这可能导致数据损坏，不仅仅是丢失事务。

设置 `innodb_flush_log_at_trx_commit` 为不为 1 的值可能导致丢失事务。然而，如果不在意持久性（ACID 中的 D），那么设置为其他的值也是有用的。也许你只是想拥有 InnoDB 的其他一些功能，例如聚簇索引、防止数据损坏，以及行锁。但仅仅因为性能原因用 InnoDB 替换 MyISAM 的情况也并不少见。

高性能事务处理需要的最佳配置是把 `innodb_flush_log_at_trx_commit` 设置为 1 且把日志文件放到一个有电池保护的写缓存的 RAID 卷中。这兼顾了安全和速度。事实上，我们敢说任何希望能扛过高负荷工作负载的产品数据库服务器，都需要有这种类型的硬件。

Percona Server 扩展了 `innodb_flush_log_at_trx_commit` 变量，使得它成为一个会话级变量，而不是一个全局变量。这允许有不同的性能和持久化要求的应用，可以使用同样的

注 14：我们说的是基于旋转盘片的机械磁盘，不是 SSD 盘，它们的性能特点完全不一样。

数据库，同时又避免了标准 MySQL 提供的一刀切的解决方案。

InnoDB 怎样打开和刷新日志以及数据文件

使用 innodb_flush_method 选项可以配置 InnoDB 如何跟文件系统相互作用。从名字来看，会以为只能影响 InnoDB 怎么写数据，实际上还影响了 InnoDB 怎么读数据。Windows 和非 Windows 的操作系统对这个选项的值是互斥的：async_unbuffered、unbuffered 和 normal 只能在 Windows 下使用，并且 Windows 下不能使用其他的值。在 Windows 下默认值是 unbuffered，其他操作系统都是 fdatasync。（如果 SHOW GLOBAL VARIABLES 显示这个变量为空，意味着它被设置为默认值了。）

> 改变 InnoDB 执行 I/O 操作的方式可以显著地影响性能，所以请确认你明白了在做什么后再去做改动！

这是个有点难以理解的选项，因为它既影响日志文件，也影响数据文件，而且有时候对
362 不同类型的文件的处理也不一样。如果有一个选项来配置日志，另一个选项来配置数据文件，这样最好了，但实际上它们混合在同一个配置项中。

下面是一些可能的值：

fdatasync

: 这在非 Windows 系统上是默认值：InnoDB 用 fsync() 来刷新数据和日志文件。

InnoDB 通常用 fsync() 代替 fdatasync()，即使这个值似乎表达的是相反的意思。fdatasync() 跟 fsync() 相似，但是只刷新文件的数据，而不包括元数据（最后修改时间，等等）。因此，fsync() 会导致更多的 I/O。然而 InnoDB 的开发者都很保守，他们发现某些场景下 fdatasync() 会导致数据损坏。InnoDB 决定了哪些方法可以更安全地使用，有一些是编译时设置的，也有一些是运行时设置的。它使用尽可能最快的安全方法。

使用 fsync() 的缺点是操作系统至少会在自己的缓存中缓冲一些数据。理论上，这种双重缓冲是浪费的，因为 InnoDB 管理自己的缓冲比操作系统能做的更加智能。然而，最后的影响跟操作系统和文件系统非常相关。如果能让文件系统做更智能的 I/O 调度和批量操作，双重缓冲可能并不是坏事。有的文件系统和操作系统可以积累写操作后合并执行，通过对 I/O 重新排序来提升效率，或者并发写入多个设备。它们也可能做预读优化，例如，若连续请求了几个顺序的块，它会通知硬盘预读下一个块。

有时这些优化有帮助，有时没有。如果你好奇你的系统中的 fsync() 会做哪些具体的事，可以阅读系统的帮助手册，看下 fsync(2)。

innodb_file_per_table 选项会导致每个文件独立地做 fsync()，这意味着写多个表不能合并到一个 I/O 操作。这可能导致 InnoDB 执行更多的 fsync() 操作。

O_DIRECT

InnoDB 对数据文件使用 O_DIRECT 标记或 directio() 函数，这依赖于操作系统。这个设置并不影响日志文件并且不是在所有的类 UNIX 系统上都有效。但至少 GNU/Linux、FreeBSD，以及 Solaris（5.0 以后的新版本）是支持的。不像 O_DSYNC 标记，它同时会影响读和写。

这个设置依然使用 fsync() 来刷新文件到磁盘，但是会通知操作系统不要缓存数据，也不要用预读。这个选项完全关闭了操作系统缓存，并且使所有的读和写都直接通过存储设备，避免了双重缓冲。

在大部分系统上，这个实现用 fcntl() 调用来设置文件描述符的 O_DIRECT 标记，所以可以阅读 fcntl(2) 的手册页来了解系统上这个函数的细节。在 Solaris 系统，这个选项用 directio()。

如果 RAID 卡支持预读，这个设置不会关闭 RAID 卡的预读[注 15]。这个设置只能关闭操作系统和文件系统的预读。

如果使用 O_DIRECT 选项，通常需要带有写缓存的 RAID 卡，并且设置为 Write-Back 策略[注 16]，因为这是典型的唯一能保持好性能的方法。当 InnoDB 和实际存储设备之间没有缓冲时使用 O_DIRECT，例如当 RAID 卡没有写缓存时，可能导致严重的性能下降。现在有了多个写线程，这个问题稍微小一点（并且 MySQL 5.5 提供了原生异步 I/O），但是通常还是有问题。 363

这个选项可能导致服务器预热时间变长，特别是操作系统的缓存很大的时候。也可能导致小容量的缓冲池（例如，默认大小的缓冲池）比缓冲 I/O（Buffered IO）方式操作要慢的多。这是因为操作系统不会通过保持更多数据在自己的缓存中来“帮助”（提升性能）。如果需要的数据不在缓冲池，InnoDB 将不得不直接从磁盘读取。

这个选项不会对 innodb_file_per_table 产生任何额外的损失。相反，如果不用 innodb_file_per_table，当使用 O_DIRECT 时，可能由于一些顺序 I/O 而遭受性能损失。这种情况的发生是因为一些文件系统（包括 Linux 所有的 *ext* 文件系统）每个 inode 有一个 Mutex。当在这些文件系统上使用 O_DIRECT 时，确实需要打开 innodb_file_per_table。我们下一章会更深入地探究文件系统。

ALL_O_DIRECT

这个选项在 Percona Server 和 MariaDB 中可用。它使得服务器在打开日志文件时，也能使用标准 MySQL 中打开数据文件的方式（O_DIRECT）。

注 15： RAID 卡的预读控制必须在 RAID 卡的设置中调整。——译者注

注 16： 就是写入会在 RAID 卡缓存上进行缓冲，不直接写到硬盘。——译者注

O_DSYNC

这个选项使日志文件调用 open() 函数时设置 O_SYNC 标记。它使得所有的写同步——换个说法，只有数据写到磁盘后写操作才返回。这个选项不影响数据文件。

O_SYNC 标记和 O_DIRECT 标记的不同之处在于 O_SYNC 没有禁用操作系统层的缓存。因此，它没有避免双重缓冲，并且它没有使写操作直接操作到磁盘。用了 O_SYNC 标记，在缓存中写数据，然后发送到磁盘。

使用 O_SYNC 标记做同步写操作，听起来可能跟 fsync() 做的事情非常相似，但是它们两个的实现无论在操作系统层还是在硬件层都非常不同。用了 O_SYNC 标记后，操作系统可能把"使用同步 I/O"标记下传给硬件层，告诉设备不要使用缓存。另一方面，fsync() 告诉操作系统把修改过的缓冲数据刷写到设备上，如果设备支持，紧接着会传递一个指令给设备刷新它自己的缓存，所以，毫无疑问，数据肯定记录在了物理媒介上。另一个不同是，用了 O_SYNC 的话，每个 write() 或 pwrite() 操作都会

364 在函数完成之前把数据同步到磁盘，完成前函数调用是阻塞的。相对来看，不用 O_SYNC 标记的写入调用 fsync() 允许写操作积累在缓存（使得每个写更快），然后一次性刷新所有的数据。

再一次吐槽下这个名称，这个选项设置 O_SYNC 标记，不是 O_DSYNC 标记，因为 InnoDB 开发者发现了 O_DSYNC 的 Bug。O_SYNC 和 O_DSYNC 类似于 fysnc() 和 fdatasync()：O_SYNC 同时同步数据和元数据，但是 O_DSYNC 只同步数据。

async_unbuffered

这是 Windows 下的默认值。这个选项让 InnoDB 对大部分写使用没有缓冲的 I/O；例外是当 innodb_flush_log_at_trx_commit 设置为 2 的时候，对日志文件使用缓冲 I/O。

这个选项使得 InnoDB 在 Windows 2000、XP，以及更新版本中对数据读写都使用操作系统的原生异步（重叠的）I/O。在更老的 Windows 版本中，InnoDB 使用自己用多线程模拟的异步 I/O。

unbuffered

只对 Windows 有效。这个选项与 async_unbuffered 类似，但是不使用原生异步 I/O。

normal

只对 Windows 有效。这个选项让 InnoDB 不要使用原生异步 I/O 或者无缓冲 I/O。

Nosync 和 littlesync

只为开发使用。这两个选项在文档中没有并且对生产环境来说不安全，不应该使用这个。

如果这些看起来像是一堆不带建议的说明，那么下面是一些建议：如果使用类 UNIX 操作系统并且 RAID 控制器带有电池保护的写缓存，我们建议使用 O_DIRECT。如果不是这样，默认值或者 O_DIRECT 都可能是最好的选择，具体要看应用类型。

InnoDB 表空间

InnoDB 把数据保存在表空间内，本质上是一个由一个或多个磁盘文件组成的虚拟文件系统。InnoDB 用表空间实现很多功能，并不只是存储表和索引。它还保存了回滚日志(旧版本行)、插入缓冲（Insert Buffer)、双写缓冲（Doublewrite Buffer，后面的章节里就会描述)，以及其他内部数据结构。

配置表空间。通过 innodb_data_file_path 配置项可以定制表空间文件。这些文件都放在 innodb_data_home_dir 指定的目录下。这是一个例子：

```
innodb_data_home_dir  = /var/lib/mysql/
innodb_data_file_path = ibdata1:1G;ibdata2:1G;ibdata3:1G
```

这里在三个文件中创建了 3GB 的表空间。有时人们并不清楚可以使用多个文件分散驱动器的负载，像这样： 365

```
innodb_data_file_path = /disk1/ibdata1:1G;/disk2/ibdata2:1G;...
```

在这个例子中，表空间文件确实放在代表不同驱动器的不同目录中，InnoDB 把这些文件首尾相连组合起来。因此，通常这种方式并不能获得太多收益。InnoDB 先填满第一个文件，当第一个文件满了再用第二个，如此循环；负载并没有真的按照希望的高性能方式分布。用 RAID 控制器是分布负载更聪明的方式。

为了允许表空间在超过了分配的空间时还能增长，可以像这样配置最后一个文件自动扩展：

```
...ibdata3:1G:autoextend
```

默认的行为是创建单个 10MB 的自动扩展文件。如果让文件可以自动扩展，那么最好给表空间大小设置一个上限，别让它扩展得太大，因为一旦扩展了，就不能收缩回来。例如，下面的例子限制了自动扩展文件最多到 2GB：

```
...ibdata3:1G:autoextend:max:2G
```

管理一个单独的表空间可能有点麻烦，尤其是如果它是自动扩展的，并且希望回收空间时（因为这个原因，我们建议关闭自动扩展功能，至少设置一个合理的空间范围）。回收空间唯一的方式是导出数据，关闭 MySQL，删除所有文件，修改配置，重启，让 InnoDB 创建新的数据文件，然后导入数据。InnoDB 这种表空间管理方式很让人头疼——不能简单地删除文件或者改变大小。如果表空间损坏了，InnoDB 会拒绝启动。对日志文件也一样的严格。如果像 MyISAM 一样随便移动文件，千万要谨慎！

innodb_file_per_table 选项让 InnoDB 为每张表使用一个文件，MySQL 4.1 和之后的版本都支持。它在数据字典存储为“表名 .ibd”的数据。这使得删除一张表时回收空间简单多了，并且可以容易地分散表到不同的磁盘上。然而，把数据放到多个文件，总体来说可能导致更多的空间浪费，因为把单个 InnoDB 表空间的内部碎片浪费分布到了多个 *.ibd* 文件。对于非常小的表，这个问题更大，因为 InnoDB 的页大小是 16 KB。即使表只有 1 KB 的数据，仍然需要至少 16 KB 的磁盘空间。

即使打开 innodb_file_per_table 选项，依然需要为回滚日志和其他系统数据创建共享表空间。没有把所有数据存在其中是明智的做法，但最好还是关闭它的自动增长，因为无法在不重新导入全部数据的情况下给共享表空间瘦身。

一些人喜欢使用 innodb_file_per_table，只是因为特别容易管理，并且可以看到每个表的文件。例如，可以通过查看文件的大小来确认表的大小，这比用 SHOW TABLE STATUS 来看快多了，这个命令需要执行很多复杂的工作来判断给一个表分配了多少页面。

366 设置 innodb_file_per_table 也有不好的一面：更差的 DROP TABLE 性能。这可能足以导致显而易见的服务器端阻塞。因为有如下两个原因：

- 删除表需要从文件系统层去掉（删除）文件，这可能在某些文件系统（ext3，说的就是你）上会很慢。可以通过欺骗文件系统来缩短这个过程：把 *.ibd* 文件链接到一个 0 字节的文件，然后手动删除这个文件，而不用等待 MySQL 来做。
- 当打开这个选项，每张表都在 InnoDB 中使用自己的表空间。结果是，移除表空间实际上需要 InnoDB 锁定和扫描缓冲池，查找属于这个表空间的页面，在一个有庞大的缓冲池的服务器上做这个操作是非常慢的。如果打算删除很多 InnoDB 表（包括临时表）并且用了 innodb_file_per_table，可能会从 Percona Server 包含的一个修复中获益，它可以让服务器慢慢地清理掉属于被删除表的页面。只需要设置 innodb_lazy_drop_table 这个选项。

什么是最终的建议？我们建议使用 innodb_file_per_table 并且给共享表空间设置大小范围，这样可以过得舒服点（不用处理那些空间回收的事）。如果遇到任何头痛的场景，就像上面说的，考虑用下 Percona 的那个修复。

提醒一下，事实上没有必要把 InnoDB 文件放在传统的文件系统上。像许多的传统数据库服务器一样，InnoDB 提供使用裸设备的选项——例如，一个没有格式化的分区——作为它的存储。然而，今天的文件系统已经可以存放足够大的文件，所以已经没有必要使用这个选项。使用裸设备可能提升几个百分点的性能，但是我们不认为这点小提升足以抵消这样做带来的坏处，我们不能直接用文件管理数据。当把数据存在一个裸设备分

区时，不能使用 *mv*、*cp* 或其他任何工具来操作它。最终，这点小的性能收益显然不值得。

行的旧版本和表空间 在一个写压力大的环境下，InnoDB 的表空间可能增长得非常大。如果事务保持打开状态很久（即使它们没有做任何事），并且使用默认的 REPEATABLE READ 事务隔离级别，InnoDB 将不能删除旧的行版本，因为没提交的事务依然需要看到它们。InnoDB 把旧版本存在共享表空间，所以如果有更多的数据在更新，共享表空间会持续增长。有时这个问题并非是没提交的事务的原因，也可能是工作负载的问题：清理过程只有一个线程处理，直到最近的 MySQL 版本才改进，这可能导致清理线程处理速度跟不上旧版本行数增加的速度。

无论发生何种情况，SHOW INNODB STATUS 的数据都可以帮助定位问题。查看历史链表的长度会显示了回滚日志的大小，以页为单位。

分析 TRANSACTIONS 部分的第一行和第二行可以证实这个观点，这部分展示了当前事务号以及清理线程完成到了哪个点。如果这个差距很大，可能有大量的没有清理的事务。 367

这有个例子：

```
------------
TRANSACTIONS
------------
Trx id counter 0 80157601
Purge done for trx's n:o <0 80154573 undo n:o <0 0
```

事务标识是一个 64 比特的数字，由两个 32 比特的数字（在更新版本的 InnoDB 中这是个十六进制的数字）组成，所以需要做一点数学计算来计算差距。在这个例子中就很简单了，因为最高位是 0：那么有 80 157 601 – 80 154 573 = 3 028 个"潜在的"没有被清理的事务（*innotop* 可以做这个计算）。我们说"潜在的"，是因为这跟有很多没有清理的行是有很大区别的。只有改变了数据的事务才会创建旧版本的行，但是有很多事务并没有修改数据（相反的，一个事务也可能修改很多行）。

如果有个很大的回滚日志并且表空间因此增长很快，可以强制 MySQL 减速来使 InnoDB 的清理线程可以跟得上。这听起来不怎么样，但是没办法。否则，InnoDB 将保持数据写入，填充磁盘直到最后磁盘空间爆满，或者表空间大于定义的上限。

为了控制写入速度，可以设置 innodb_max_purge_lag 变量为一个大于 0 的值。这个值表示 InnoDB 开始延迟后面的语句更新数据之前，可以等待被清除的最大的事务数量。你必须知道工作负载以决定一个合理的值。例如，事务平均影响 1KB 的行，并且可以容许表空间里有 100MB 的未清理行，那么可以设置这个值为 100 000。

牢记，没有清理的行版本会对所有的查询产生影响，因为它们事实上使得表和索引更大

了。如果清理线程确实跟不上，性能可能显著的下降。设置 innodb_max_purge_lag 变量也会降低性能，但是它的伤害较少。[注17]

在更新版本的 MySQL 中，甚至在更早版本的 Percona Server 和 MariaDB，清理过程已经显著地提升了性能，并且从其他内部工作任务中分离出来。甚至可以创建多个专用的清理线程来更快地做这个后台工作。如果可以利用这些特性，会比限制服务器的服务能力要好得多。

368

双写缓冲（Doublewrite Buffer）

InnoDB 用双写缓冲来避免页没写完整所导致的数据损坏。当一个磁盘写操作不能完整地完成时，不完整的页写入就可能发生，16KB 的页可能只有一部分被写到磁盘上。有多种多样的原因（崩溃、Bug，等等）可能导致页没有写完整。双写缓冲在这种情况发生时可以保证数据完整性。

双写缓冲是表空间一个特殊的保留区域，在一些连续的块中足够保存 100 个页。本质上是一个最近写回的页面的备份拷贝。当 InnoDB 从缓冲池刷新页面到磁盘时，首先把它们写（或者刷新）到双写缓冲，然后再把它们写到其所属的数据区域中。这可以保证每个页面的写入都是原子并且持久化的。

这意味着每个页都要写两遍？是的，但是因为 InnoDB 写页面到双写缓冲是顺序的，并且只调用一次 fsync() 刷新到磁盘，所以实际上对性能的冲击是比较小的——通常只有几个百分点，肯定没有一半那么多，尽管这个开销在 SSD 上更明显，我们下一章会讨论这个问题。更重要的是，这个策略允许日志文件更加高效。因为双写缓冲给了 InnoDB 一个非常牢固的保证，数据页不会损坏，InnoDB 日志记录没必要包含整个页，它们更像是页面的二进制变化量。

如果有一个不完整的页写到了双写缓冲，原始的页依然会在磁盘上它的真实位置。当 InnoDB 恢复时，它将用原始页面替换掉双写缓冲中的损坏页面。然而，如果双写缓冲成功写入，但写到页的真实位置失败了，InnoDB 在恢复时将使用双写缓冲中的拷贝来替换。InnoDB 知道什么时候页面损坏了，因为每个页面在末尾都有校验值（Checksum）。校验值是最后写到页面的东西，所以如果页面的内容跟校验值不匹配，说明这个页面是损坏的。因此，在恢复的时候，InnoDB 只需要读取双写缓冲中每个页面并且验证校验值。如果一个页面的校验值不对，就从它的原始位置读取这个页面。

有些场景下，双写缓冲确实没必要——例如，你也许想在备库上禁止双写缓冲。此外一些文件系统（例如 ZFS）做了同样的事，所以没必要再让 InnoDB 做一遍。可以通过设

注 17： 请注意，这种实现思路是一个存在很多争议的话题，请看 MySQL 的 bug 60776 来获得更多细节信息。

置 innodb_doublewrite 为 0 来关闭双写缓冲。在 Percona Server 中，可以配置双写缓冲存到独立的文件中，所以可以把这部分工作压力分离出来放在单独的盘上。

其他的 I/O 配置项

sync_binlog 选项控制 MySQL 怎么刷新二进制日志到磁盘。默认值是 0，意味着 MySQL 并不刷新，由操作系统自己决定什么时候刷新缓存到持久化设备。如果这个值比 0 大，它指定了两次刷新到磁盘的动作之间间隔多少次二进制日志写操作（如果 autocommit 被设置了，每个独立的语句都是一次写，否则就是一个事务一次写）。把它设置为 0 和 1 369
以外的值是很罕见的。

如果没有设置 sync_binlog 为 1，那么崩溃以后可能导致二进制日志没有同步事务数据。这可以轻易地导致复制中断，并且使得及时恢复变得不可能。无论如何，可以把这个值设置为 1 来获得安全的保障。这样就会要求 MySQL 同步把二进制日志和事务日志这两个文件刷新到两个不同的位置。这可能需要磁盘寻道，相对来说是个很慢的操作。

像 InnoDB 日志文件一样，把二进制日志放到一个带有电池保护的写缓存的 RAID 卷，可以极大地提升性能。事实上，写和刷新二进制日志缓存其实比 InnoDB 事务日志要昂贵多了，因为不像 InnoDB 事务日志，每次写二进制日志都会增加它们的大小。这需要每次写入文件系统都更新元信息。所以，设置 sync_binlog=1 可能比 innodb_flush_log_at_trx_commit=1 对性能的损害要大得多，尤其是网络文件系统，例如 NFS。

一个跟性能无关的提示，关于二进制日志：如果希望使用 expire_logs_days 选项来自动清理旧的二进制日志，就不要用 *rm* 命令去删。服务器会感到困惑并且拒绝自动删除它们，并且 PURGE MASTER LOGS 也将停止工作。解决的办法是，如果发现了这种情况，就手动重新同步“主机名 -bin.index”文件，可以用磁盘上现有日志文件的列表来更新。

我们将在下一章更深入地涉及 RAID，但是值得在这里重复一下，把带有电池保护写缓存的高质量 RAID 控制器设置为使用写回（Writeback）策略，可以支持每秒数千的写入，并且依然会保证写到持久化存储。数据写到了带有电池的高速缓存，所以即使系统断电它也能存在。但电源恢复时，RAID 控制器会在磁盘被设置为可用前，把数据从缓存中写到磁盘。因此，一个带有电池保护写缓存的 RAID 控制器可以显著地提升性能，这是非常值得的投资。当然，SSD 存储是另一个选择，我们也会在下一章讲到。

8.5.2 MyISAM 的 I/O 配置

让我们从分析 MyISAM 怎么为索引操作 I/O 开始。MyISAM 通常每次写操作之后就把索引变更刷新磁盘。如你打算在一张表上做很多修改，那么毫无疑问，批量操作会更快一些。

一种办法是用 LOCK TABLES 延迟写入，直到解锁这些表。这是个提升性能的很有价值的技巧，因为它使得你精确控制哪些写被延迟，以及什么时候把它们刷到磁盘。可以精确延迟那些希望延迟的语句。

370 通过设置 delay_key_write 变量，也可以延迟索引的写入。如果这么做，修改的键缓冲块直到表被关闭才会刷新。[注18] 可能的配置如下：

OFF

MyISAM 每次写操作后刷新键缓冲（键缓存，Key Buffer）中的脏块到磁盘，除非表被 LOCK TABLES 锁定了。

ON

打开延迟键写入，但是只对用 DELAY_KEY_WRITE 选项创建的表有效。

ALL

所有的 MyISAM 表都会使用延迟键写入。

延迟键写入在某些场景下可能很有帮助，但是通常不会带来很大的性能提升。当键缓冲的读命中很好但写命中不好时，数据又比较小，这可能很有用。当然也有一小部分缺点：

- 如果服务器缓存并且块没有被刷到磁盘，索引可能会损坏。
- 如果很多写被延迟了，MySQL 可能需要花费更长时间去关闭表，因为必须等待缓冲刷新到磁盘。在 MySQL 5.0 这可能引起很长的表缓存锁。
- 由于上面提到的原因，FLUSH TABLES 可能需要很长时间。如果为了做逻辑卷（LVM）快照或者其他备份操作，而执行 FLUSH TABLES WITH READ LOCK，那可能增加操作的时间。
- 键缓冲中没有刷回去的脏块可能占用空间，导致从磁盘上读取的新块没有空间存放。因此，查询语句可能需要等待 MyISAM 释放一些键缓存的空间。

另外，除了配置 MyISAM 的索引 I/O 还可以配置 MyISAM 怎样尝试从损坏中恢复。myisam_recover 选项控制 MyISAM 怎样寻找和修复错误。需要在配置文件或者命令行中设置这个选项。可以通过下面的 SQL 语句查看选项的值，但是不能修改（这不是个印刷错误——系统里变量名跟命令的变量名有差异）：

```
mysql> SHOW VARIABLES LIKE 'myisam_recover_options';
```

打开这个选项通知 MySQL 在表打开时，检查是否损坏，并且在找到问题的时候进行修复。可以设置的值如下：

注 18： 表可能因为多种原因被关闭。例如，服务器因为表缓存没有空间了就会关闭表，或者有人执行了 FLUSH TABLES。

DEFAULT（或者不设置） 371

使 MySQL 尝试修复任何被标记为崩溃或者没有标记为完全关闭的表。默认值不要求在恢复时执行其他动作。跟大多数变量不同，这里 DEFAULT 值不是重置变量的值为编译值；它本质上意味着"没有设置"。

BACKUP

让 MySQL 将数据文件的备份写到 *.BAK* 文件，以便随后进行检查。

FORCE

即使 *.MYD* 文件中丢失的数据可能超过一行，也让恢复继续。

QUICK

除非有删除块，否则跳过恢复。块中有已经删除的行也依然会占用空间，但是可以被后面的 INSERT 语句重用。这可能比较有用，因为 MyISAM 大表的恢复可能花费相当长的时间。

可以使用多个设置，用逗号分隔。例如"BACKUP,FORCE"会强制恢复并且创建备份。这是为什么我们在这一章前面部分的示例配置中这么用的原因。

我们建议打开这个选项，尤其是只有一些小的 MyISAM 表时。服务器运行着一些损坏的 MyISAM 表是很危险的，因为它们有时可以导致更多数据损坏，甚至服务器崩溃。然而，如果有很大的表，原子恢复是不切实际的：它导致服务器打开所有的 MyISAM 表时都会检查和修复，这是低效的做法。在这段时间，MySQL 会阻止连接做任何工作。如果有一大堆的 MyISAM 表，比较好的主意还是启动后用 CHECK TABLES 和 REPAIR TABLES 命令来做[注19]，这样对服务器影响比较少。不管哪种方式，检查和修复表都是很重要的。

打开数据文件的内存映射（MMAP）访问是另一个有用的 MyISAM 选项。内存映射使得 MyISAM 直接通过操作系统的页面缓存访问 *.MYD* 文件，避免系统调用的开销。在 MySQL 5.1 和更新的版本中，可以通过 myisam_use_mmap 选项打开内存映射。更老版本的 MySQL 只能对压缩的 MyISAM 表使用内存映射。

8.6 配置 MySQL 并发

当 MySQL 承受高并发压力时，可能会遇到不曾遇到过的瓶颈。这个章节阐述了当这些问题出现的时候，怎样去发现它们，以及在 MyISAM 和 InnoDB 遇到这样的压力时怎样 372 获得尽可能最好的性能。

注 19：一些 Debian 系统会自动做这些事，像一个钟摆的摆动，朝着不同的方向不停地摇摆。只是把这个行为配置为 Debian 默认做的事不是一个好主意，应该由 DBA 来决定。

8.6.1 InnoDB 并发配置

InnoDB 是为高性能设计的，在最近几年它的提升非常明显，但依然不完美。InnoDB 架构在有限的内存、单 CPU、单磁盘的系统中仍然暴露出一些根本性问题。在高并发场景下，InnoDB 的某些方面的性能可能会降低，唯一的办法是限制并发。可以参考第 3 章中使用的技巧来诊断并发问题。

如果在 InnoDB 并发方面有问题，解决方案通常是升级服务器。相比当前的版本，像 MySQL 5.0 和早期的 MySQL 5.1 这样的旧版本，在高并发下完全是个悲剧。所有的东西都在全局 Mutex（例如，缓冲池 Mutex）上排队，导致服务器几乎陷入停顿。如果升级到某个更新版本的 MySQL，在大部分场景都不再需要限制并发。

如果需要这么做，这里会介绍它是怎么工作的。InnoDB 有自己的“线程调度器”控制线程怎么进入内核访问数据，以及它们在内核中一次可以做哪些事。最基本的限制并发的方式是使用 innodb_thread_concurrency 变量，它会限制一次性可以有多少线程进入内核，0 表示不限制。如果在旧的 MySQL 版本里有 InnoDB 并发问题，这个变量是最重要的配置之一[注20]。

在任何架构和业务压力下，给这个变量设置个“靠谱”的值都很重要，理论上，下面的公式可以给出一个这样的值 ：

```
并发值 = CPU 数量 * 磁盘数量 *2
```

但是在实践中，使用更小的值会更好一点。必须做实验来找出适合系统的最好的值。

如果已经进入内核的线程超过了允许的数量，新的线程就无法再进入内核。InnoDB 使用两段处理来尝试让线程尽可能高效地进入内核。两段策略减少了因操作系统调度引起的上下文切换。线程第一次休眠 innodb_thread_sleep_delay 微秒，然后再重试。如果它依然不能进入内核，则放入一个等待线程队列，让操作系统来处理。

373 第一阶段默认的休眠时间是 10 000 微秒。当 CPU 有大量的线程处在“进入队列前的休眠”状态，因而没有被充分利用时，改变这个值在高并发环境里可能会有帮助。如果有大量的小查询，默认值可能也太大了，因为这增加了 10 毫秒的查询延时。

一旦线程进入内核，它会有一定数量的“票据（Tickets)”，可以让它“免费”返回内核，不需再做并发检查。这限制了一个线程回到其他等待线程之前可以做多少事。innodb_concurrency_tickets 选项控制票据的数量。它很少需要修改，除非有很多运行时间极长的查询。票据是按查询授权的，不是按事务。一旦查询完成，它没用完的票据就销毁了。

注 20： 事实上，在某些工作负载下，并发限制实现可能自己就成为了系统的瓶颈，所以有时它需要打开，但另一些时候它需要关闭。性能分析会告诉你该怎么做。

除了缓冲池和其他结构的瓶颈，还有另一个提交阶段的并发瓶颈，这个时候 I/O 非常密集，因为需要做刷新操作。innodb_commit_concurrency 变量控制有多少个线程可以在同一时间提交。如果 innodb_thread_concurrency 配置得很低也有大量的线程冲突，那么配置这个选项可能会有帮助。

最后，有一个新的解决方案值得考虑：使用线程池（Thread Pool）来限制并发。原始的线程池实现已经随着 MySQL 6.0 的代码树一起被废弃了，并且有严重缺陷。但是 MariaDB 已经重新实现了，并且 Oracle 最近放出了一个商业插件可以为 MySQL 5.5 提供线程池功能。对这些东西我们都没有足够的经验来指导你怎么做，你也许会更加困惑，因为我们会指出这两种实现似乎都不满足 Facebook，它在自己内部私有的 MySQL 分支中有一个叫做“准入控制”的特殊功能。如果可能的话，在这本书的第 4 版我们将分享一些线程池的知识，以及什么时候它们可以工作，什么时候不能工作。

8.6.2 MyISAM 并发配置

在某些条件下，MyISAM 也允许并发插入和读取，这使得可以“调度”某些操作以尽可能少地产生阻塞。

在讲述 MyISAM 的并发设置之前，理解 MyISAM 是怎样删除和插入行的，是非常重要的。删除操作不会重新整理整个表，它们只是把行标记为删除，在表中留下“空洞”。MyISAM 倾向于在可能的时候填满这些空洞，在插入行时重新利用这些空间。如果没有空洞了，它就把新行插入表的末尾。

尽管 MyISAM 是表级锁，它依然可以一边读取，一边并发追加新行。这种情况下只能读取到查询开始时的所有数据，新插入的数据是不可见的。这样可以避免不一致读。

然而，若表中间的某些数据变动了的话，还是难以提供一致读。MVCC 是解决这个问题最流行的方法：一旦修改者创建了新版本，它就让读取者读数据的旧版本。可是，MyISAM 并不像 InnoDB 那样支持 MVCC，所以除非插入操作在表的末尾，否则不能支持并发插入。 374

通过设置 concurrent_insert 这个变量，可以配置 MyISAM 打开并发插入，可以配置为如下值：

`0`

MyISAM 不允许并发插入，所有插入都会对表加互斥锁。

`1`

这是默认值。只要表中没有空洞，MyISAM 就允许并发插入。

2

这个值在 MySQL 5.0 以及更新版本中有效。它强制并发插入到表的末尾，即使表中有空洞。如果没有线程从表中读取数据，MySQL 将把新行放在空洞里。使用这个设置通常会使表更加碎片化。

如果合并操作可以更加高效，也可以配置 MySQL 对一些操作进行延迟。举个实例，可以通过 delay_key_write 变量延迟写索引，正如这一章前面我们提到的。这牵涉到熟悉的权衡：立即写索引（安全但是昂贵），或者等待但是祈求在写发生前别断电（更快，但是遇到崩溃时可能引起巨大的索引损坏，因为索引文件已经过期了）。

也可以让 INSERT、REPLACE、DELETE、以及 UPDATE 语句的优先级比 SELECT 语句更低，设置 low_priority_updates 选项就可以了。这相当于把 LOW_PRIORITY 修饰符应用到全局 UPDATE 语句。当使用 MyISAM 时，这是个非常重要的选项，这让 SELECT 语句可以获得相当好的并发度，否则一小部分获取高优先级写锁的语句就可能导致 SELECT 无法获取资源。

最后，尽管 InnoDB 的扩展性问题更经常被提及，但是 MyISAM 一样也有长时间获取 Mutex 的问题。在 MySQL 4.0 和更早版本里，有一个全局的 Mutex 保护所有的键缓存 I/O，在多处理器和多磁盘环境下很容易引起扩展性问题。MySQL 4.1 的键缓存代码做了改进，就不再有这些问题了，但是它依然对每个键缓冲区持有一个 Mutex。当一个线程从键缓冲中复制键数据块到本地磁盘时会有竞争，从磁盘上读取时就没这个问题。磁盘瓶颈没了，但是当你在键缓冲里访问数据时，另一个瓶颈出现了。有时可以围绕这个问题把键缓冲分成多个区，但是这条路不总是行得通。例如，只涉及一个独立索引的时候，这问题就没有办法解决。于是，在多处理器的机器上 SELECT 查询并发可能相对单 CPU 的机器显著下降，即使当时只有这些 SELECT 查询在执行。

MariaDB 提供分开的（分区的）键缓冲，如果经常遇到这个问题，也许可以带来帮助。

375 8.7 基于工作负载的配置

配置服务器的一个目标是把它定制得符合特定的工作负载。这需要精通所有类型的服务器活动的数量、类型，以及频率——不仅仅是查询语句，也包括其他的活动，例如连接服务器以及刷新表。

第一件应该做的事情是熟悉你的服务器，如果还没做就赶紧。了解什么样的查询跑在上面。用例如 *innotop* 这样的工具来监控它，用 *pt-query-digest* 来创建查询报告。这不仅帮助你全面地了解服务器正在做什么，还可以知道查询花费大量时间做了哪些事。第 3 章阐明了怎么把这些东西找出来。

当服务器在满载情况下运行时，请尝试记录所有的查询语句，因为这是最好的方式来查看哪种类型的查询语句占用资源最多。同时，创建 processlist 快照，通过 state 或者 command 字段来聚合它们（*innotop* 可以实现，或者可以使用第 3 章展示的脚本）。例如，是否大量地在复制数据到临时表，或者排序数据？如果有，也许需要优化查询语句，以及查看临时表和排序缓冲配置项。

8.7.1 优化 BLOB 和 TEXT 的场景

BLOB 和 TEXT 列对 MySQL 来说是特殊类型的场景（我们把所有 BLOB 和 TEXT 都简单称为 BLOB 类型，因为它们属于相同类型的数据）。BLOB 值有几个限制使得服务器对它的处理跟其他类型不一样。一个最重要的注意事项是，服务器不能在内存临时表中存储 BLOB 值[注 21]，因此，如果一个查询涉及 BLOB 值，又需要使用临时表——不管它多小——它都会立即在磁盘上创建临时表。这样效率很低，尤其是对小而快的查询。临时表可能是查询中最大的开销。

有两种办法来减轻这个不利的情况：通过 SUBSTRING() 函数（第 4 章有更多关于这个函数的细节）把值转换为 VARCHAR，或者让临时表更快一些。

让临时表运行更快的最好方式是，把它们放在基于内存的文件系统（GNU/Linux 上是 *tmpfs*）。这会降低一些开销，尽管这依然比内存表慢许多。因为操作系统会避免把数据写到磁盘，所以内存文件系统可以帮助提升性能[注 22]。一般的文件系统也会在内存中缓存，但是操作系统会每隔几秒就刷新一次。*tmpfs* 文件系统从来不会刷新，它就是为低开销和简单起见而设计的。例如，没必要为这个文件系统预备任何恢复方案。这使得它更快。 376

服务器设置里控制临时表文件放在哪的是 tmpdir。建议监控文件系统使用率以保证有足够的空间存放临时表。如果需要，可以指定多个临时表存放位置，MySQL 将会轮询使用。

如果 BLOB 列非常大，并且用的是 InnoDB，也许可以调大 InnoDB 日志缓冲大小。在这一章前面有更多关于这方面的内容。

对于很长的变长列（例如，BLOB、TEXT，以及长字符列），InnoDB 存储一个 768 字节的前缀在行内[注 23]。如果列的值比前缀长，InnoDB 会在行外分配扩展存储空间来存剩下的部分。它会分配一个完整的 16KB 的页，像其他所有的 InnoDB 页面一样，每个列都有自己的页面（不同的列不会共享扩展存储空间）。InnoDB 一次只为一个列分配一个页的扩

注 21：最近版本的 Percona Server 对某些场景消除了这个限制。

注 22：如果操作系统把它交换（Swap）出内存，数据依然会到磁盘。

注 23：这个长度足够在列上创建一个 255 字符的索引，即使是 utf8 的（每个字符可能需要三个字节）。前缀是 InnoDB 的 Antelope 文件格式特有的，MySQL 5.1 和更新版本中的 Barracuda 格式（默认不打开的）没有前缀。

展存储空间，直到使用了超过 32 个页以后，就会一次性分配 64 个页面。

注意，我们说过 InnoDB 可能会分配扩展存储空间。如果总的行长（包括大字段的完整长度）比 InnoDB 的最大行长限制要短（比 8KB 小一些），InnoDB 将不会分配扩展存储空间，即使大字段（Long column）的长度超过了前缀长度。

最后，当 InnoDB 更新存储在扩展存储空间中的大字段时，将不会在原来的位置更新。而是会在扩展存储空间中写一个新值到一个新的位置，并且不会删除旧的值。

所有这一切都有以下后果：

- 大字段在 InnoDB 里可能浪费大量空间。例如，若存储字段值只是比行的要求多了一个字节，也会使用整个页面来存储剩下的字节，浪费了页面的大部分空间。同样的，如果有一个值只是稍微超过了 32 个页的大小，实际上就需要使用 96 个页面。
- 扩展存储禁用了自适应哈希，因为需要完整地比较列的整个长度，才能发现是不是正确的数据（哈希帮助 InnoDB 非常快速地找到"猜测的位置"，但是必须检查"猜测的位置"是不是正确）。因为自适应哈希是完全的内存结构，并且直接指向 Buffer Pool 中访问"最"频繁的页面，但对于扩展存储空间却无法使用自适应哈希。
- 太长的值可能使得在查询中作为 WHERE 条件不能使用索引，因而执行很慢。在应用 WHERE 条件之前，MySQL 需要把所有的列读出来，所以可能导致 MySQL 要求 InnoDB 读取很多扩展存储，然后检查 WHERE 条件，丢弃所有不需要的数据。查询不
377 需要的列绝不是好主意，在这种特殊的场景下尤其需要避免这样做。如果发现查询正遇到这个限制带来的问题，可以尝试通过覆盖索引来解决部分问题。
- 如果一张表里有很多大字段，最好是把它们组合起来单独存到一个列里面，比如说用 XML 文档格式存储。这让所有的大字段共享一个扩展存储空间，这比每个字段用自己的页要好。
- 有时候可以把大字段用 COMPRESS() 压缩后再存为 BLOB，或者在发送到 MySQL 前在应用程序中进行压缩，这可以获得显著的空间优势和性能收益。

8.7.2 优化排序（Filesorts）

从第 6 章我们知道 MySQL 有两种排序算法。如果查询中所有需要的列和 ORDER BY 的列总大小超过 max_length_for_sort_data 字节，则采用 two-pass 算法。或者当任何需要的列——即使没有被 ORDER BY 使用的列——是 BLOB 或者 TEXT，也会采用这个算法。（可以用 SUBSTRING() 把这些列转换一下，就可以用 single-pass 算法了。）

MySQL 有两个变量可以控制排序怎样执行。通过修改 max_length_for_sort_data 变

量[注24]的值，可以影响 MySQL 选择哪种排序算法。因为 single-pass 算法为每行需要排序的数据创建一个固定大小的缓冲，对于 VARCHAR 列，在和 max_length_for_sort_data 比较时，使用的是其定义的最大长度，而不是所存储数据的实际长度。这也是为什么我们建议只选择必要的列的一个原因。

当 MySQL 必须排序 BLOB 或 TEXT 字段时，它只会使用前缀，然后忽略剩下部分的值。这是因为缓冲只能分配固定大小的结构体来保存要排序的值，然后从扩展存储空间中复制前缀到这个结构体中。使用 max_sort_length 变量可以指定这个前缀有多大。

可惜，MySQL 无法查看它用了哪个算法。如果增加了 max_length_for_sort_data 变量的值，磁盘使用率上升了，CPU 使用率下降了，并且 Sort_merge_passes 状态变量相对于修改之前开始很快地上升，也许是强制让很多的排序使用了 single-pass 算法。

8.8 完成基本配置 378

我们已经完成了服务器内核的旅程——希望你喜欢这个旅程！现在让我们回到示例配置，并且看下怎样修改剩下的配置。

我们已经讨论了怎样设置一般的选项，例如数据目录、InnoDB 和 MyISAM 缓存、日志，还有其他的一些。让我们重温剩下的那些：

tmp_table_size 和 max_heap_table_size

这两个设置控制使用 Memory 引擎的内存临时表能使用多大的内存。如果隐式内存临时表的大小超过这两个设置的值，将会被转换为磁盘 MyISAM 表，所以它的大小可以继续增长。(隐式临时表是一种并非由自己创建，而是服务器创建，用于保存执行中的查询的中间结果的表。)

应该简单地把这两个变量设为同样的值。我们的示例配置文件中选择了 32M。这可能不够，但是要谨防这个变量太大了。临时表最好呆在内存里，但是如果它们被撑得很大，实际上还是让它们使用磁盘比较好，否则可能会让服务器内存溢出。

假设查询语句没有创建庞大的临时表(通常可以通过合理的索引和查询设计来避免)，那把这些变量设大一点，免得需要把内存临时表转换为磁盘临时表。这个过程可以在 SHOW PROCESSLIST 中看到。

可以查看服务器的 SHOW STATUS 计数器在某段时间内的变化，以此来查看创建临时表的频率以及是否是磁盘临时表。你不能判断一张（临时）表是先创建为内存表然后被转换为了磁盘表，还是一开始就创建的磁盘表（可能因为有 BLOB 字段），但

注 24：在带有 LIMIT 语句的查询中，MySQL 5.6 会改变排序缓冲的用法，并且会修正一个可导致执行一个昂贵的安装历程而使用庞大的排序缓冲的问题，所以如果升级到了 MySQL 5.6，需要特别小心地检查这些设置中任何自定义的设置。

是至少可以看到创建磁盘临时表有多频繁。仔细检查 Created_tmp_disk_tables 和 Created_tmp_tables 变量。

max_connections

这个设置的作用就像一个紧急刹车，以保证服务器不会因应用程序激增的连接而不堪重负。如果应用程序有问题，或者服务器遇到如连接延迟的问题，会创建很多新连接。但是如果不能执行查询，那打开一个连接没有好处，所以被“太多的连接”的错误拒绝是一种快速而代价小的失败方式。

把 max_connections 设置得足够高，以容纳正常可能达到的负载，并且要足够安全，能保证允许你登录和管理服务器。例如，若认为正常情况将有 300 或者更多连接，
379
则可以设置为 500 或者更多。如果不知道将会有多少连接，500 也不是一个不合理的起点。默认值是 100，对大部分应用来说这都不够。

要时时小心可能遇到连接限制的突然袭击。例如，若重新启动应用服务器，可能没有把它的连接关闭干净，同时 MySQL 可能没有意识到它们已经被关闭了。当应用服务器重新开始运转，并试图打开到数据库的连接，就可能由于挂起的连接还没有超时，而使新连接被拒绝。

观察 Max_used_connections 状态变量随着时间的变化。这个是高水位标记，可以告诉你服务器连接是不是在某个时间点有个尖峰。如果这个值达到了 max_connections，说明客户端至少被拒绝了一次，并且当它重现的时候，应该使用第 3 章中的技巧来抓取服务器的活动状态。

thread_cache_size

设置这个变量，可以通过观察服务器一段时间的活动，来计算一个有理有据的值。观察 Threads_connected 状态变量并且找到它在一般情况下的最大值和最小值。你也许希望把线程缓存设置得足够大，以在高峰和低谷时都足够，甚至可能更大方一些，因为就算设置得有点太大了，一般也不是大问题。你也许可以设置为波动范围两到三倍的大小。例如，若 Threads_connected 状态从 150 变化到 175，可以设置线程缓存为 75。但是也不用设置得非常大，因为保持大量等待连接的空闲线程并没有什么真正的用处。250 的上限是个不错的估算值（或者 256，如果你喜欢 2 的次方。）

也可以观察 Threads_created 状态随着时间的变化。如果这个值很大或者一直增长，这是另一个线索，告诉你可能需要调大 thread_cache_size 变量。查看 Threads_cached 来看有多少线程已经在缓存中了。

一个相关的状态变量是 Slow_launch_threads。这个状态如果是个很大的值，那么意味着某些情况延迟了连接分配新线程。这也是个线索，可能服务器有些问题了，但是不能明确地指出是哪出问题了。一般来说，可能是系统过载了，导致操作系统不能为新创建的线程调度 CPU。这不是说你就需要增加线程缓存的大小了。你应该诊断这个问题并且修复它，而不是用缓存来掩盖问题，因为这还可能导致其他问题。

table_cache_size

这个缓存（或者在 MySQL 5.1 中被分成两个缓存区）应该被设置得足够大，以避免总是需要重新打开和重新解析表的定义。你可以通过观察 Open_tables 的值及其在一段时间的变化来检查该变量。如果你看到 Opened_tables 每秒变化很大，那么 table_cache 值可能不够大。隐式临时表也可能导致打开表的数量不断增长，即使表缓存并没有用满，所以这可能也没什么问题。 380

该问题的线索应该是 Opened_tables 不断地增长，即使 Open_tables 并不跟 table_cache_size 一样大。

虽然表缓存很有用，也不应该把这个变量设置得太大。表缓存可能在两种情况下适得其反。

首先，MySQL 没有一个很有效的方法来检查缓存，所以如果真的太大了，可能效率会下降。在大部分情况下，不应该把它设置得大于 10 000，或者是 10 240，如果喜欢使用 2 的 *N* 次方的话。[注 25]

第二个原因是有些类型的工作负载是不能缓存的。如果工作负载不是可缓存的，不管把缓存设置得多大，任何访问都无法在缓存命中，忘记缓存吧，把它设置为 0！这可以避免情况变得更糟糕，缓存不命中比昂贵的缓存检查后再不命中还是要好的。什么类型的工作负载不是可缓存的？如果有几万或几十万张表，并且它们都很均匀地被使用，就不可能把它们全缓存了，最好把这个变量设得小一点。当系统上有数量非常多的并行应用而其中没有一个是非常忙碌的，有时候这是适当的。

这个值从 max_connections 的 10 倍开始设置是比较有道理的，但是再次说明，在大部分场景下最好保持在 10 000 以下甚至更低。

还有其他一些类型的设置可能经常会包含在配置文件中，包括二进制日志以及复制设置。二进制日志对恢复到某个时间点，以及复制是非常有用的，另外复制还有一些它自己的设置。我们会在本书后面的章节中覆盖复制和备份的重要设置。

8.9 安全和稳定的设置

基本配置设置到位后，可能希望启用一些使服务器更安全和更可靠的设置。它们中的一些会影响性能，因为保证安全性和可靠性往往要付出一些代价。有些人意识到了：他们能阻止愚蠢的错误发生，比如把无意义的数据插入服务器，以及一些变动在日常操作中没有啥区别，只是在很边缘的情况防止糟糕的事情发生。

让我们首先来看看收集的一些对一般服务器都有用的配置项：

注 25： 你听说过一个关于二进制的笑话嘛？世界上有 10 种人：部分是懂二进制的，部分不懂二进制。还有另外 10 种人：一些认为二进制 / 十进制的笑话有意思，一些是精虫上脑。我们不会说我们是否认为这是滑稽的。

381 expire_logs_days

如果启用了二进制日志，应该打开这个选项，可以让服务器在指定的天数之后清理旧的二进制日志。如果不启用,最终服务器的空间会被耗尽,导致服务器卡住或崩溃。我们建议把这个选项设置得足够从两个备份之前恢复(在最近的备份失败的情况下)。即使每天都做备份，还是建议留下 7 ~ 14 天的二进制日志。从我们的经验来看，当遇到一些不常见的问题时，你会感谢有这一两个星期的二进制日志。例如重搭一个备机再次尝试赶上主库。应该保持足够多的二进制日志，遇到这些情况时可以给自己一些呼吸的空间。

max_allowed_packet

这个设置防止服务器发送太大的包，也会控制多大的包可以被接收。默认值可能太小了，但设置得太大也可能有危险。如果设置得太小，有时复制上会出问题，通常表现为备库不能接收主库发过来的复制数据。你也许需要增加这个设置到 16MB 或者更大。这些文档里没有，但这个选项也控制在一个用户定义的变量的最大值，所以如果需要非常大的变量，要小心——如果超过这个变量的大小，它们可能被截断或者设置为 NULL。

max_connect_errors

如果有时网络短暂抽风了，或者应用配置出现错误，或者有另外的问题，如权限，在短暂的时间内不断地尝试连接，客户端可能被列入黑名单，然后将无法连接，直到再次刷新主机缓存。这个选项的默认设置太小了，很容易导致问题。你也许希望增加这个值，实际上，如果知道服务器可以充分抵御蛮力攻击，可以把这个值设得非常大，以有效地禁用主机黑名单。

skip_name_resolve

这个选项禁用了另一个网络相关和鉴权认证相关的陷阱：DNS 查找。DNS 是 MySQL 连接过程中的一个薄弱环节。当连接服务器时，默认情况下，它试图确定连接和使用的主机的主机名，作为身份验证凭据的一部分。(就是说，你的凭据是用户名，主机名、以及密码——并不只是用户名和密码）但是验证主机来源，服务器需要执行 DNS 的正向和反向查找。要是 DNS 有问题就悲剧了，在某些时间点这是必然的事。当发生这样的情况时，所有事都会堆积起来，最终导致连接超时。为了避免这种情况，我们强烈建议设置这个选项，在验证时关闭 DNS 查找。然而，如果这
382 么做,需要把基于主机名的授权改为用 IP 地址、通配符,或者特定主机名“localhost”,因为基于主机名的账号会被禁用。

sql_mode

这个设置可以接受多种多样的值来改变服务器行为。我们不建议只是为了好玩而改变这个值；最好在大多数情况下让 MySQL 像 MySQL，不要尝试让它的行为像其他数据库服务器。(许多客户端和图形界面工具，除了 MySQL 还有它们自己的 SQL

方言，例如，若修改它用更符合 ANSI 的 SQL，有些操作会没法做。）然而，有些选项值是很有用的，有些在具体情况可能是值得考虑的。建议查看文档中下面这些选项，并且考虑使用它们：STRICT_TRANS_TABLES、ERROR_FOR_DIVISION_BY_ZERO、NO_AUTO_CREATE_USER、NO_AUTO_VALUE_ON_ZERO、NO_ENGINE_SUBSTITUTION、NO_ZERO_DATE、NO_ZERO_IN_DATE 和 ONLY_FULL_GROUP_BY。

然而，要意识到对已经存在的应用修改这些设置值可不是个好主意，因为这么做可能让服务器跟应用预期不兼容。人们不经意间写的查询中应用的列不在 GROUP BY 中，或者使用聚合函数，这种情况非常常见，例如，若想打开 ONLY_FULL_GROUP_BY 选项，最好首先在开发或未上线服务器上做一下测试，一旦要在生产环境部署则必须确认所有地方都可以工作。

sysdate_is_now

这是另一个可能导致与应用预期向后不兼容的选项。但如果不是明确需要 SYSDATE() 函数的非确定性行为（非确定性行为可能会导致复制中断或者使得基于时间点的备份恢复结果不可信），那么你可能希望打开该选项以确保 SYSDATE() 函数有确定的行为。

下面的选项可以控制复制行为，并且对防止备库出问题非常有帮助：

read_only

这个选项禁止没有特权的用户在备库做变更，只接受从主库传输过来的变更，不接受从应用来的变更。我们强烈建议把备库设置为只读模式。

skip_slave_start

这个选项阻止 MySQL 试图自动启动复制。因为在不安全的崩溃或其他问题后，启动复制是不安全的，所以需要禁用自动启动，用户需要手动检查服务器，并确定它是安全的之后再开始复制。

slave_net_timeout

这个选项控制备库发现跟主库的连接已经失败并且需要重连之前等待的时间。默认值是一个小时，太长了。设置为一分钟或更短。

sync_master_info、sync_relay_log、sync_relay_log_info 383

这些选项，在 MySQL 5.5 以及更新版本中可用，解决了复制中备库长期存在的问题：不把它们的状态文件同步到磁盘，所以服务器崩溃后可能需要人来猜测复制的位置实际上在主库是哪个位置，并且可能在中继日志（Relay Log）里有损坏。这些选项使得备库崩溃后，更容易从崩溃中恢复。这些选项默认是不打开的，因为它们会导致备库额外的 fsync() 操作，可能会降低性能。如果有很好的硬件，我们建议打开这些选项，如果复制中出现 fsync() 造成的延时问题，就应该关闭它们。

Percona Server 中有一种侵入性更小的方式来做这些工作，即打开 innodb_

overwrite_relay_log_info 选项。这可以让 InnoDB 在事务日志中存储复制的位置，这是完全事务化的，并且不需要任何额外的 fsync() 操作。在崩溃恢复期间，InnoDB 会检查复制的元信息文件，如果文件过期了就更新为正确的位置。

8.10 高级 InnoDB 设置

回到第 1 章我们讨论的 InnoDB 历史：首先是内建（built-in）的版本，然后有了两个有效版本，现在更新的版本再次变成了一个。更新的 InnoDB 代码有更多的功能和非常好的扩展性。如果正在使用 MySQL 5.1，应该明确地配置 MySQL 忽略旧版本的 InnoDB 而使用新版的。这将极大地提升服务器性能。需要打开 ignore_builtin_innodb 选项，然后配置 plugin_load 选项把 InnoDB 作为插件打开。建议参考 InnoDB 文档中对应平台上的扩展语法[注26]。

对于新版本的 InnoDB，有一些新的选项可以用。如果启用，它们中有些对服务器性能相当重要，也有一些安全性和稳定性的选项，如下所示。

innodb

这个看似平淡无奇的选项实际上非常重要，如果把这个值设置为 FORCE，只有在 InnoDB 可以启动时，服务器才会启动。如果使用 InnoDB 作为默认存储引擎，这一定是你期望的结果。你应该不会希望在 InnoDB 失败（例如因为错误的配置而导致的不可启动）的情况下启动服务器，因为写的不好的应用可能之后会连接到服务器，导致一些无法预知的损失和混乱。最好是整个服务器都失败，强制你必须查看错误日志，而不是以为服务器正常启动了。

384 innodb_autoinc_lock_mode

这个选项控制 InnoDB 如何生成自增主键值，某些情况下，例如高并发插入时，自增主键可能是个瓶颈。如果有很多事务等待自增锁（可以在 SHOW ENGINE INNODB STATUS 里看到），应该审视这个变量的设置。手册上已经详细解释了该选项的行为，在此我们就不再重复了。

innodb_buffer_pool_instances

这个选项在 MySQL 5.5 和更新的版本中出现，可以把缓冲池切分为多段，这可能是在高负载的多核机器上提升 MySQL 可扩展性最重要的一个方式了。多个缓冲池分散了工作压力，所以一些全局 Mutex 竞争就没有那么大了。

目前尚不清楚什么情况下应该选择多个缓冲池实例。我们运行过八个实例的基准，但是直到 MySQL 5.5 已经广泛部署了很长一段时间，我们依然不明白多个缓冲池实

注 26：在 Percona Server 中，只有一个版本的 InnoDB，并且是内建的，所以你不需要禁用一个版本然后载入另一个版本替换它。

例的一些微妙之处。

我们不是暗示 MySQL 5.5 没有在生产环境广泛部署。只是对我们已经帮助解决过的大部分互斥锁相互争用的极端场景的用户来说，升级可能需要很多个月的时间来计划、验证，并执行。这些用户有时运行着高度定制化的 MySQL 版本，使得更加倍谨慎地对待升级。当越来越多的这类用户升级到 MySQL 5.5，并以他们独特的方式进行压力验证，我们可能会学到关于多缓冲池的一些我们没见过的有趣的事情。也许直到那时，我们才可以说运行八个缓冲池实例是非常有益的。

值得注意的是 Percona Server 用了不同的方法来解决 InnoDB 互斥锁争用问题。相对于把缓冲池分成多个——一个在许多像 InnoDB 的系统下经过检验无可否认的方法——我们选择把一些全局 Mutex 拆分为更细、更专用的 Mutex。我们的测试显示最好的方式是结合这两种方法，在 Percona Server 5.5 版本中已经可用了：多缓冲区和更细粒度的锁。

`innodb_io_capacity`

InnoDB 曾经在代码里写死了假设服务器运行在每秒 100 个 I/O 操作的单硬盘上。默认值很糟糕。现在可以告诉 InnoDB 服务器有多大的 I/O 能力。InnoDB 有时需要把这个设置得相当高（在像 PCI-E SSD 这样极快的存储设备上需要设置为上万）才能稳定地刷新脏页，原因解释起来相当复杂。

`innodb_read_io_threads` 和 `innodb_write_io_threads` 385

这些选项控制有多少后台线程可以被 I/O 操作使用。最近版本的 MySQL 里，默认值是 4 个读线程和 4 个写线程，对大部分服务器这都足够了，尤其是 MySQL 5.5 里面可以用操作系统原生的异步 I/O 以后。如果有很多硬盘并且工作负载并发很大，可以发现这些线程很难跟上，这种情况下可以增加线程数，或者可以简单地把这个选项的值设置为可以提供 I/O 能力的磁盘数量（即使后面是一个 RAID 控制器）。

`innodb_strict_mode`

这个设置让 MySQL 在某些条件下把警告改成抛错，尤其是无效的或者可能有风险的 `CREATE TABLE` 选项。如果打开这个设置，就必然会检查所有 `CREATE TABLE` 选项，因为它不会让你创建一些用起来比较爽（但是有隐患）的表。有时这有点悲观，过于严格了。当尝试恢复备份时可能就不希望打开这个选项了。

`innodb_old_blocks_time`

InnoDB 有个两段缓冲池 LRU（最近最少使用）链表，设计目的是防止换出长期使用很多次的页面。像 *mysqldump* 产生的这种一次性的（大）查询，通常会读取页面到缓冲池的 LRU 列表，从中读取需要的行，然后移动到下一页。理论上，两段 LRU 链表将阻止此页取代很长一段时间内都需要用到的页面被放入“年轻（Young）”子链表，并且只在它已被浏览过多次后将其移动到“年老（Old）”子链表。但是 InnoDB 默认没有配置为防止这种情况，因为页内有很多行，所以从页面读取的行的

多次访问，会导致它立即被转移到“年老（Old）”子链表，对那些需要长时间缓存的页面带来换出的压力。

这个变量指定一个页面从LRU链表的“年轻”部分转移到“年老”部分之前必须经过的毫秒数。默认情况下它设置为0，将它设为诸如1 000毫秒（一秒）这样的小一点的值，在我们的基准测试中已被证明非常有效。

8.11 总结

在阅读完这一章节之后，你应该有了一个比默认设置好得多的服务器配置。服务器应该更快更稳定了，并且除非运行出现了罕见的状况，都应该没有必要再去做优化配置的工作了。

复习一下，我们建议从参考示例配置文件开始，设置符合服务器和工作负载的基本选项，增加安全性和完整性所需的选项，并且，如果合适的话，在MySQL 5.5中配置新版的InnoDB Plugin才有的配置项。这就是关于优化服务器配置所需要做的全部的事情。

386 如果使用的是InnoDB，最重要的选项是下面这两个：

- `innodb_buffer_pool_size`
- `innodb_log_file_size`

恭喜你——你解决了我们见过的真实存在的配置问题中的绝大部分！如果使用我们的在线配置工具 *http://tools.percona.com*，对这些问题和其他配置选项的使用，会得到很好的建议。

我们也提出了很多关于不要做什么的建议。其中最重要的是不要“调优”服务器；不要使用比率、公式或“调优脚本”作为设置配置变量的基础；不要信任来自互联网上的不明身份的人的意见；不要为了看起来很糟糕的事情去不断地刷`SHOW STATUS`。如果有些设置其实是错误的，在剖析服务器性能时也会展现出来。

有几个重要的设置没有在本章讨论，主要是因为它们是为特定类型的硬件和工作负载服务的。我们暂不讨论这些设置，因为我们相信，任何关于怎样设置的意见，都需要与内部流程的解释工作一起来做。这给我们带来了下一章，它会告诉你如何优化MySQL的硬件和操作系统，反之亦然。

第 9 章

操作系统和硬件优化

MySQL 服务器性能受制于整个系统最薄弱的环节，承载它的操作系统和硬件往往是限制因素。磁盘大小、可用内存和 CPU 资源、网络，以及所有连接它们的组件，都会限制系统的最终容量。因此，需要小心地选择硬件，并对硬件和操作系统进行合适的配置。例如，若工作负载是 I/O 密集型的，一种方法是设计应用程序使得最大限度地减少 MySQL 的 I/O 操作。然而，更聪明的方式通常是升级 I/O 子系统，安装更多的内存，或重新配置现有的磁盘。

硬件的更新换代非常迅速，所以本章有关特定产品或组件的内容可能将很快变得过时。像往常一样，我们的目标是帮助提升对这些概念的理解，这样对于即使没有直接覆盖到的知识也可以举一反三。这里我们将通过现有的硬件来阐明我们的观点。

9.1 什么限制了 MySQL 的性能

许多不同的硬件都可以影响 MySQL 的性能，但我们认为最常见的两个瓶颈是 CPU 和 I/O 资源。当数据可以放在内存中或者可以从磁盘中以足够快的速度读取时，CPU 可能出现瓶颈。把大量的数据集完全放到大容量的内存中，以现在的硬件条件完全是可行的[注1]。

另一方面，I/O 瓶颈，一般发生在工作所需的数据远远超过有效内存容量的时候。如果应用程序是分布在网络上的，或者如果有大量的查询和低延迟的要求，瓶颈可能转移到网络上，而不再是磁盘 I/O[注2]。

注 1： 普通 PC Server 也能配到 192GB 内存。——译者注
注 2： 网络吞吐也是一种 I/O。——译者注

388 第 3 章中提及的技巧可以帮助找到系统的限制因素，但即使你认为已经找到了瓶颈，也应该透过表象去看更深层次的问题。某一方面的缺陷常常会将压力施加在另一个子系统，导致这个子系统出问题。例如，若没有足够的内存，MySQL 可能必须刷出缓存来腾出空间给需要的数据——然后，过了一小会，再读回刚刚刷新的数据（读取和写入操作都可能发生这个问题）。本来是内存不足，却导致出现了 I/O 容量不足。当找到一个限制系统性能的因素时，应该问问自己，"是这个部分本身的问题，还是系统中其他不合理的压力转移到这里所导致的？"在第 3 章的诊断案例中也有讨论到这个问题。

还有另外一个例子：内存总线的瓶颈也可能表现为 CPU 问题。事实上，我们说一个应用程序有"CPU 瓶颈"或者是"CPU 密集型"，真正的意思应该是计算的瓶颈。接下来将深入探讨这个问题。

9.2 如何为 MySQL 选择 CPU

在升级当前硬件或购买新的硬件时，应该考虑下工作负载是不是 CPU 密集型。

可以通过检查 CPU 利用率来判断是否是 CPU 密集型的工作负载，但是仅看 CPU 整体的负载是不合理的，还需要看看 CPU 使用率和大多数重要的查询的 I/O 之间的平衡，并注意 CPU 负载是否分配均匀。本章稍后讨论的工具可以用来弄清楚是什么限制了服务器的性能。

9.2.1 哪个更好：更快的 CPU 还是更多的 CPU

当遇到 CPU 密集型的工作时，MySQL 通常可以从更快的 CPU 中获益（相对更多的 CPU）。

但这不是绝对的，因为还依赖于负载情况和 CPU 数量。更古老的 MySQL 版本在多 CPU 上有扩展性问题，即使新版本也不能对单个查询并发利用多个 CPU。因此，CPU 速度限制了每个 CPU 密集型查询的响应时间。

当我们讨论 CPU 的时候，为保证本文易于阅读，对某些术语将不会做严格的定义。现在一般的服务器通常都有多个插槽（Socket），每个插槽上都可以插一个有多个核心的 CPU（有独立的执行单元），并且每个核心可能有多个"硬件线程"。这些复杂的架构需要有点耐心去了解，并且我们不会总是明确地区分它们。不过，在一般情况下，当谈到 CPU 速度的时候，谈论的其实是执行单元的速度，当提到的 CPU 数量时，指的通常是在操作系统上看到的数量，尽管这可能是独立的执行单元数量的多倍[注3]。

注 3： 超线程技术。——译者注

这几年 CPU 在各个方面都有了很大的提升。例如，今天的 Intel CPU 速度远远超过前几代，这得益于像直接内存连接（directly attached memory）技术以及 PCIe 卡之类的设备互联上的改善等。这些改进对于存储设备尤其有效，例如 Fusion-io 和 Virident 的 PCIe 闪存驱动器。 389

超线程的效果相比以前也要好得多，现在操作系统也更了解如何更好地使用超线程。而以前版本的操作系统无法识别两个虚拟处理器实际上是在同一芯片上，认为它们是独立的，于是会把任务安排在两个实际上是相同物理执行单元上的虚拟处理器。实际上单个执行单元并不是真的可以在同一时间运行两个进程，所以这样做会发生冲突和争夺资源。而同时其他 CPU 却可能在闲置，从而浪费资源。操作系统需要能感知超线程，因为它必须知道什么时候执行单元实际上是闲置的，然后切换相应的任务去执行。这个问题之前常见的原因是在等待内存总线，可能花费需要高达一百个 CPU 周期，这已经类似于一个轻量级的 I/O 等待。新的操作系统在这方面有了很大的改善。超线程现在已经工作得很好。过去，我们时常提醒人们禁用它，但现在已经不需要这样做了。

这就是说，现在可以得到大量的快速的 CPU——比本书的第 2 版出版的时候要多得多。所以多和快哪个更重要？一般来说两个都想要。从广义上来说，调优服务器可能有如下两个目标：

低延时（快速响应）

要做到这一点，需要高速 CPU，因为每个查询只能使用一个 CPU。

高吞吐

如果能同时运行很多查询语句，则可以从多个 CPU 处理查询中受益。然而，在实践中，还要取决于具体情况。因为 MySQL 还不能在多个 CPU 中完美地扩展，能用多少个 CPU 还是有极限的。在旧版本的 MySQL 中（MySQL 5.1 以后的版本已经有一些提升），这个限制非常严重。在新的版本中，则可以放心地扩展到 16 或 24 个 CPU，或者更多，取决于使用的是哪个版本（Percona 往往在这方面略占优势）。

如果有多路 CPU，并且没有并发执行查询语句，MySQL 依然可以利用额外的 CPU 为后台任务（例如清理 InnoDB 缓冲、网络操作，等等）服务。然而，这些任务通常比执行查询语句更加轻量化。

MySQL 复制（将在下一章中讨论）也能在高速 CPU 下工作得非常好，而多 CPU 对复制的帮助却不大。如果工作负载是 CPU 密集型，主库上的并发任务传递到备库以后会被简化为串行任务，这样即使备库硬件比主库好，也可能无法保持跟主库之间的同步。也就是说，备库的瓶颈通常是 I/O 子系统，而不是 CPU。

如果有一个 CPU 密集型的工作负载，考虑是需要更快的 CPU 还是更多 CPU 的另外一个 390

因素是查询语句实际在做什么。在硬件层面，一个查询可以在执行或等待。处于等待状态常见的原因是在运行队列中等待（进程已经是可运行状态，但所有的CPU都忙）、等待闩锁（Latch）或锁（Lock）、等待磁盘或网络。那么你期望查询是等待什么呢？如果等待闩锁或锁，通常需要更快的CPU；如果在运行队列中等待，那么更多或者更快的CPU都可能有帮助。（也可能有例外，例如，查询等待InnoDB日志缓冲区的Mutex，直到I/O完成前都不会释放——这可能表明需要更多的I/O容量）。

这就是说，MySQL在某些工作负载下可以有效地利用很多CPU。例如，假设有很多连接查询的是不同表（假设这些查询不会造成表锁的竞争，实际上对MyISAM和MEMORY表可能会有问题），并且服务器的总吞吐量比任何单个查询的响应时间都更重要。吞吐量在这种情况下可以非常高，因为线程可以同时运行而互不争用。

再次说明，在理论上这可能更好地工作：不管查询是读取不同的表还是相同的表，InnoDB都会有一些全局共享的数据结构，而MyISAM在每个缓冲区都有全局锁。而且不仅仅是存储引擎，服务器层也有全局锁。以前InnoDB承担了所有的骂名，但最近做了一些改进后，暴露了服务器层中的其他瓶颈。例如臭名昭著的`LOCK_open`互斥量（Mutex），在MySQL 5.1和更早版本中可能就是个大问题，另外还有其他一些服务器级别的互斥量（例如查询缓存）。

通常可以通过堆栈跟踪来诊断这些类型的竞争问题，例如Percona Toolkit中的*pt-pmp*工具。如果遇到这样的问题，可能需要改变服务器的配置，禁用或改变引起问题的组件，进行数据分片（Sharding），或者通过某种方式改变做事的方法。这里无法列举所有的问题和相应的解决方案，但是一旦有一个确定的诊断，答案通常是显而易见的。大部分不幸遇到的问题都是边缘场景，最常见的问题随着时间的推移都在服务器上被修复了。

9.2.2 CPU架构

可能99%以上的MySQL实例（不含嵌入式使用）都运行在Intel或者AMD芯片的x86架构下。本书中我们基本都是针对这种情况。

64位架构现在都是默认的了，32位CPU已经很难买到了。MySQL在64位架构上工作良好，尽管有些事暂时不能利用64位架构来做。因此，如果使用的是较老旧版本的MySQL，在64位服务器上可能要小心。例如，在MySQL 5.0发布的早期时候，每个
391 MyISAM键缓冲区被限制为4 GB，由一个32位整数负责寻址。（可以创建多个键缓冲区来解决这个问题。）

确保在64位硬件上使用64位操作系统！最近这种情况已经不太常见了，但以前经常可以遇到，大多数主机托管提供商暂时还是在服务器上安装32位操作系统，即使是64位

CPU。32 位操作系统意味着不能使用大量的内存：尽管某些 32 位系统可以支持大量的内存，但不能像 64 位系统一样有效地利用，并且在 32 位系统上，任何一个单独的进程都不能寻址 4 GB 以上的内存。

9.2.3 扩展到多个 CPU 和核心

多 CPU 在联机事务处理（OLTP）系统的场景中非常有用。这些系统通常执行许多小的操作，并且是从多个连接发起请求，因此可以在多个 CPU 上运行。在这样的环境中，并发可能成为瓶颈。大多数 Web 应用程序都属于这一类。

OLTP 服务器一般使用 InnoDB，尽管它在多 CPU 的环境中还存在一些未解决的并发问题。然而，不只是 InnoDB 可能成为瓶颈：任何共享资源都是潜在的竞争点。InnoDB 之所以获得大量关注是因为它是高并发环境下最常见的存储引擎，但 MyISAM 在大压力时的表现也不好，即使不修改任何数据只是读取数据也是如此。许多并发瓶颈，如 InnoDB 的行级锁和 MyISAM 的表锁，没有办法优化——除了尽可能快地处理任务之外，没有别的办法解决，这样，锁就可以尽快分配给等待的任务。如果一个锁是造成它们（其他任务）都在等待的原因，那么不管有多少 CPU 都一样。因此，即使是一些高并发工作负载，也可以从更快的 CPU 中受益。

实际上有两种类型的数据库并发问题，需要不同的方法来解决，如下所示。

逻辑并发问题

应用程序可以看到资源的竞争，如表或行锁争用。这些问题通常需要好的策略来解决，如改变应用程序、使用不同的存储引擎、改变服务器的配置，或使用不同的锁定提示或事务隔离级别。

内部并发问题

比如信号量、访问 InnoDB 缓冲池页面的资源争用，等等。可以尝试通过改变服务器的设置、改变操作系统，或使用不同的硬件解决这些问题，但通常只能缓解而无法彻底消灭。在某些情况下，使用不同的存储引擎或给存储引擎打补丁，可以帮助缓解这些问题。

MySQL 的"扩展模式"是指它可以有效利用的 CPU 数量，以及在压力不断增长的情况下如何扩展，这同时取决于工作负载和系统架构。通过"系统架构"的手段是指通过调整操作系统和硬件，而不是通过优化使用 MySQL 的应用程序。CPU 架构（RISC、CISC、流水线深度等）、CPU 型号和操作系统都影响 MySQL 的扩展模式。这也是为什么说基准测试是非常重要的：一些系统可以在不断增加的并发下依然运行得很好，而另一些的表现则糟糕得多。 392

有些系统在更多的处理器下甚至可能降低整体性能。这是相当普遍的情况，我们了解到许多人试图升级到有多个 CPU 的系统，最后只能被迫恢复到旧系统（或绑定 MySQL 进程到其中某些核心），因为这种升级反而降低了性能。在 MySQL 5.0 时代，Google 的补丁和 Percona Server 出现之前，能有效利用的 CPU 核数是 4 核，但是现在甚至可以看到操作系统报告多达 80 个"CPU"的服务器。如果规划一个大的升级，必须要同时考虑硬件、服务器版本和工作负载。

某些 MySQL 扩展性瓶颈在服务器层，而其他一些在存储引擎层。存储引擎是怎么设计的至关重要，有时更换到一个不同的引擎就可以从多处理器上获得更多效果。

我们看到在世纪之交围绕处理器速度的战争在一定程度上已经平息，CPU 厂商更多地专注于多核 CPU 和多线程的变化。CPU 设计的未来很可能是数百个处理器核心，四核心和六核心的 CPU 在今天是很常见的。不同厂商的内部架构差异很大，不可能概括出线程、CPU 和内核之间的相互作用。内存和总线如何设计也是非常重要的。归根结底，多个内核和多个物理 CPU 哪个更好，这是由硬件体系结构决定的。

现代 CPU 的另外两个复杂之处也值得提一下。首先是频率调整。这是一种电源管理技术，可以根据 CPU 上的压力而动态地改变 CPU 的时钟速度。问题是，它有时不能很好地处理间歇性突发的短查询的情况，因为操作系统可能需要一段时间来决定 CPU 的时钟是否应该变化。结果，查询可能会有一段时间速度较慢，并且响应时间增加了。频率调整可能使间歇性的工作负载性能低下，但可能更重要的是，它会导致性能波动。

第二个复杂之处是 boost 技术，这个技术改变了我们对 CPU 模式的看法。我们曾经以为四核 2GHz CPU 有四个同样强大的核心，不管其中有些是闲置或非闲置。因此，一个完美的可扩展系统，当它使用所有四个内核的时候，可以预计得到四倍的提升。但是现在已经不是这样了，因为当系统只使用一个核心时，处理器会运行在更高的时钟速度上，例如 3GHz。这给很多的规划容量和可扩展性建模的工具出了一个难题，因为系统性能表现不再是线性的变化了。这也意味着，"空闲 CPU"并不代表相同规模的资源浪费，如果有一台服务器上只运行了备库的复制，而复制执行是单线程的，所以有三个 CPU 是
393 空闲的，因此认为可以利用这些 CPU 资源执行其他任务而不影响复制，可能就想错了。

9.3 平衡内存和磁盘资源

配置大量内存最大的原因其实不是因为可以在内存中保存大量数据：最终目的是避免磁盘 I/O，因为磁盘 I/O 比在内存中访问数据要慢得多。关键是要平衡内存和磁盘的大小、速度、成本和其他因素，以便为工作负载提供高性能的表现。在讨论如何做到这一点之前，暂时先回到基础知识上来。

计算机包含一个金字塔型的缓存体系，更小、更快、更昂贵的缓存在顶端，如图 9-1 所示。

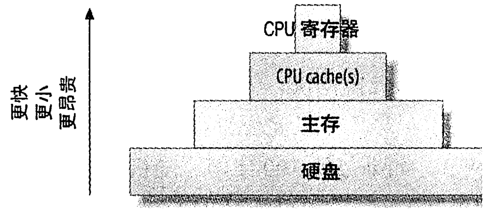


图9-1：缓存层级

在这个高速缓存层次中，最好是利用各级缓存来存放“热点”数据，以获得更快的访问速度，通常使用一些启发式的方法，例如“最近被使用的数据可能很快再次被使用”以及“相邻的数据可能很快需要使用”，这些算法非常有效，因为它们参考了空间和时间的局部性原理。

从程序员的视角来看，CPU 寄存器和高速缓存是透明的，并且与硬件架构相关。管理它们是编译器和 CPU 的工作。然而，程序员会有意识地注意到内存和硬盘的不同，并且在程序中通常区分使用它们[注4]。

在数据库服务器上尤其明显，其行为往往非常符合我们刚才提到的预测算法所做的预测。设计良好的数据库缓存（如 InnoDB 缓冲池），其效率通常超过操作系统的缓存，因为操作系统缓存是为通用任务设计的。数据库缓存更了解数据库存取数据的需求，它包含特殊用途的逻辑（例如写入顺序）以帮助满足这些需求。此外，系统调用不需要访问数据库中的缓存数据。

这些专用的缓存需求就是为什么必须平衡缓存层次结构以适应数据库服务器特定的访问模式的原因。因为寄存器和芯片上的高速缓存不是用户可配置的，内存和存储是唯一可以改变的东西。 394

9.3.1 随机 I/O 和顺序 I/O

数据库服务器同时使用顺序和随机 I/O，随机 I/O 从缓存中受益最多。想像有一个典型的混合工作负载，均衡地包含单行查找与多行范围扫描，可以说服自己相信这个说法。典型的情况是“热点”数据随机分布。因此，缓存这些数据将有助于避免昂贵的磁盘寻道。相反，顺序读取一般只需要扫描一次数据，所以缓存对它是没用的，除非能完全放在内

注 4： 然而，程序可能依赖大量在操作系统内存中缓存的数据，对程序来说，概念上属于“在磁盘上”的数据。例如，MyISAM 就是这么做的，它把数据文件放在磁盘上，并通过操作系统缓存磁盘上的数据，使其访问速度更快。

存中缓存起来。

顺序读取不能从缓存中受益的另一个原因是它们比随机读快。这有以下两个原因：

顺序 I/O 比随机 I/O 快。

顺序操作的执行速度比随机操作快，无论是在内存还是磁盘上。假设磁盘每秒可以做 100 个随机 I/O 操作，并且可以完成每秒 50MB 的顺序读取（这大概是消费级磁盘现在能达到的水平）。如果每行 100 字节，随机读每秒可以读 100 行，相比之下顺序读可以每秒读 500 000 行——是随机读的 5 000 倍，或几个数量级的差异。因此，在这种情况下随机 I/O 可以从缓存中获得很多好处。

顺序访问内存行的速度也快于随机访问。现在的内存芯片通常每秒可以随机访问约 250 000 次 100 字节的行，或者每秒 500 万次的顺序访问。请注意，内存随机访问速度比磁盘随机访问快了 2 500 倍，而内存中顺序访问只有磁盘 10 倍的速度。

存储引擎执行顺序读比随机读快。

一个随机读一般意味着存储引擎必须执行索引操作。（这个规则也有例外，但对 InnoDB 和 MyISAM 都是对的）。通常需要通过 B 树的数据结构查找，并且和其他值比较。相反，连续读取一般需要遍历一个简单的数据结构，例如链表。这样就少了很多工作，反复这样操作，连续读取的速度就比随机读取要快了。

最后，随机读取通常只要查找特定的行，但不仅仅只读取一行——而是要读取一整页的数据，其中大部分是不需要的。这浪费了很多工作。另一方面，顺序读取数据，通常发生在想要的页面上的所有行，所以更符合成本效益。

395 综上所述，通过缓存顺序读取可以节省一些工作，但缓存随机读取可以节省更多的工作。换句话说，如果能负担得起，增加内存是解决随机 I/O 读取问题最好的办法。

9.3.2 缓存，读和写

如果有足够的内存，就完全可以避免磁盘读取请求。如果所有的数据文件都可以放在内存中，一旦服务器缓存“热”起来了，所有的读操作都会在缓存命中。虽然还是会有逻辑读取，不过物理读取就没有了。但写入是不同的问题。写入可以像读一样在内存中完成，但迟早要被写入到磁盘，所以它是需要持久化的。换句话说，缓存可延缓写入，但不能像消除读取一样消除写入。

事实上，除了允许写入被延迟，缓存可以允许它们被集中操作，主要通以下两个重要途径：

多次写入，一次刷新

一片数据可以在内存中改变很多次，而不需要把所有的新值写到磁盘。当数据最终被刷新到磁盘后，最后一次物理写之前发生的修改都被持久化了。例如，许多语句

可以更新内存中的计数器。如果计数器递增 100 次，然后写入到磁盘，100 次修改就被合并为一次写。

I/O 合并

许多不同部分的数据可以在内存中修改，并且这些修改可以合并在一起，通过一次磁盘操作完成物理写入。

这就是为什么许多交易系统使用预写日志（WAL）策略。预写日志采用在内存中变更页面，而不马上刷新到磁盘上的策略，因为刷新磁盘通常需要随机 I/O，这非常慢。相反，如果把变化的记录写到一个连续的日志文件，这就很快了。后台线程可以稍后把修改的页面刷新到磁盘；并在刷新过程中优化写操作。

写入从缓冲中大大受益，因为它把随机 I/O 更多地转换到连续 I/O。异步（缓冲）写通常是由操作系统批量处理，使它们能以更优化的方式刷新到磁盘。同步（无缓冲）写必须在写入到磁盘之后才能完成。这就是为什么它们受益于 RAID 控制器中电池供电的回写（Write-Back）高速缓存（我们稍后讨论 RAID）。

9.3.3 工作集是什么

每个应用程序都有一个数据的“工作集”——就是做这个工作确实需要用到的数据。很多数据库都有大量不在工作集内的数据。

可以把数据库想象为有抽屉的办公桌。工作集就是放在桌面上的完成工作必须使用的文件。桌面是这个比喻中的主内存，而抽屉就是硬盘。 396

就像完成工作不需要办公桌里每一张纸一样，也不需要把整个数据库装到内存中来获得最佳性能——只需要工作集就可以。

工作集大小的不同取决于应用程序。对于某些应用程序，工作集可能是总数据大小的 1%，而对于其他应用，也可能接近 100%。当工作集不能全放在内存中时，数据库服务器必须在磁盘和内存之间交换数据，以完成工作。这就是为什么内存不足可能看起来却像 I/O 问题。有时没有办法把整个工作集的数据放在内存中，并且有时也并不真的想这么做（例如，若应用需要大量的顺序 I/O）。工作集能否完全放在内存中，对应用程序体系结构的设计会产生很大的影响。

工作集可以定义为基于时间的百分比。例如，一小时的工作集可能是一个小时内数据库使用的 95% 的页面，除了 5% 的最不常用的页面。百分比是考虑这个问题最有用的方式，因为每小时可能需要访问的数据只有 1%，但超过 24 小时，需要访问的数据可能会增加到整个数据库中 20% 的不同页面。根据需要被缓存起来的数据量多少，来思考工作集会

更加直观，缓存的数据越多，工作负载就越可能成为 CPU 密集型。如果不能缓存足够的数据，工作集就不能完全放在内存中。

应该依据最常用的页面集来考虑工作集，而不是最频繁读写的页面集。这意味着，确定工作集需要在应用程序内有测量的模块，而不能仅仅看外部资源的利用，例如 I/O 访问，因为页面的 I/O 操作跟逻辑访问页面不是同一回事。例如，MySQL 可能把一个页面读入内存，然后访问它数百万次，但如果查看 *strace*，只会看到一个 I/O 操作。缺乏确定工作集所需的检测模块，最大的原因是没有对这个主题有较多的研究。

工作集包括数据和索引，所以应该采用缓存单位来计数。一个缓存单位是存储引擎工作的数据最小单位。

不同存储引擎的缓存单位大小是不一样的，因此也使得工作集的大小不一样。例如，InnoDB 在默认情况下是 16 KB 的页。如果 InnoDB 做一个单行查找需要读取磁盘，就需要把包含该行的整个页面读入缓冲池进行缓存，这会引起一些缓存的浪费。假设要随机访问 100 字节的行。InnoDB 将用掉缓冲池中很多额外的内存来缓存这些行，因为每一行都必须读取和缓存一个完整的 16KB 页面。因为工作集也包括索引，InnoDB 也会读取并缓存查找行所需的索引树的一部分。InnoDB 的索引页大小也是 16 KB，这意味着访问
397 一个 100 字节的行可能一共要使用 32 KB 的缓存空间（有可能更多，这取决于索引树有多深）。因此，缓存单位也是在 InnoDB 中精心挑选聚集索引非常重要的另一个原因。聚集索引不仅可以优化磁盘访问，还可以帮助在同一页面存储相关的数据，因此在缓存中可以尽量放下整个工作集。

9.3.4 找到有效的内存 / 磁盘比例

找到一个良好的内存 / 磁盘比例最好的方式是通过试验和基准测试。如果可以把所有东西放入内存，你就大功告成了——后面没有必要再为此考虑什么。但大多数的时候不可能这么做，所以需要用数据的一个子集来做基准测试，看看将会发生什么。测试的目标是一个可接受的缓存命中率。缓存未命中是当有查询请求数据时，数据不能在内存中命中，服务器需要从磁盘获取数据。

缓存命中率实际上也会决定使用了多少 CPU，所以评估缓存命中率的最好方法是查看 CPU 使用率。例如，若 CPU 使用了 99% 的时间工作，用了 1% 的时间等待 I/O，那缓存命中率还是不错的。

让我们考虑下工作集是如何影响高速缓存命中率的。首先重要的一点，要认识到工作集不仅是一个单一的数字而是一个统计分布，并且缓存命中率是非线性分布的。例如，有

10GB 内存，并且缓存未命中率为 10%，你可能会认为只需要增加 11% 以上的内存注5，就可以降低缓存的未命中率到 0。但实际上，诸如缓存单位的大小之类的问题会导致缓存效率低下，可能意味着理论上需要 50GB 的内存，才能把未命中率降到 1%。即使与一个完美的缓存单位相匹配，理论预测也可能是错误的：例如数据访问模式的因素也可能让事情更复杂。解决 1% 的缓存未命中率甚至可能需要 500GB 的内存，这取决于具体的工作负载！

有时候很容易去优化一些可能不会带来多少好处的地方。例如，10% 的未命中率可能导致 80% 的 CPU 使用率，这已经是相当不错的了。假设增加内存，并能够让缓存未命中率下降到 5%，简单来说，将提供另外约 6% 的数据给 CPU。再简化一下，也可以说，把 CPU 使用率增加到了 84.8%。然而，考虑到为了得到这个结果需要购买的内存，这可不一定是一个大胜利。在现实中，因为内存和磁盘访问速度之间的差异、CPU 真正操作的数据，以及许多其他因素，降低缓存未命中率到 5% 可能都不会太多改变 CPU 使用率。

这就是为什么我们说，你应该争取一个可接受的缓存命中率，而不是将缓存未命中率降低到零。没有一个应该作为目标的数字，因为“可以接受”怎么定义，取决于应用程序和工作负载。有些应用程序有 1% 的缓存未命中都可以工作得非常好，而另一些应用实际上需要这个比例低到 0.01% 才能良好运转。（“良好的缓存未命中率”是个模糊的概念，其实有很多方法来进一步计算未命中率。） 398

最好的内存 / 磁盘的比例还取决于系统上的其他组件。假设有 16 GB 的内存、20 GB 的数据，以及大量未使用的磁盘空间系统。该系统在 80% 的 CPU 利用率下运行得很好。如果想在这个系统上放置两倍多的数据，并保持相同的性能水平，你可能会认为只需要让 CPU 数量和内存量也增加到两倍。然而，即使系统中的每个组件都按照增加的负载扩展相同的量（一个不切实际的假设），这依然可能会使得系统无法正常工作。有 20GB 数据的系统可能使用了某些组件超过 50% 的容量——例如，它可能已经用掉了每秒 I/O 最大操作数的 80%。并且在系统内排队也是非线性的。服务器将无法处理两倍的负载。因此，最好的内存 / 磁盘比例取决于系统中最薄弱的组件。

9.3.5 选择硬盘

如果无法满足让足够的数据在内存中的目标——例如，估计将需要 500 GB 的内存才能完全让 CPU 负载起当前的 I/O 系统——那么应该考虑一个更强大的 I/O 子系统，有时甚至要以牺牲内存为代价，同时应用程序的设计应该能处理 I/O 等待。

这听起来似乎有悖常理。毕竟，我们刚刚说过，更多的内存可以缓解 I/O 子系统的压力，

注 5： 正确的数字是 11% 而不是 10%。10% 的未命中率对应 90% 的命中率，所以你需要用 10GB 除以 90%，就是 11.111GB。

并减少 I/O 等待。为什么要加强 I/O 子系统呢，如果只增加内存能解决问题吗？答案就在所涉及的因素之间的平衡，例如读写之间的平衡，每个 I/O 操作的大小，以及每秒有多少这样的操作发生。例如，若需要快速写日志，就不能通过增加大量有效内存来避免磁盘写入。在这种情况下，投资一个高性能的 I/O 系统与带电池支持的写缓存或固态存储，可能是个更好的主意。

作为一个简要回顾，从传统磁盘读取数据的过程分为三个步骤：

1. 移动读取磁头到磁盘表面上的正确位置。
2. 等待磁盘旋转，所有所需的数据在读取磁头下。
3. 等待磁盘旋转过去，所有所需的数据都被读取磁头读出。

磁盘执行这些操作有多快，可以浓缩为两个数字：访问时间（步骤 1 和 2 合并）和传输速度。399 这两个数字也决定延迟和吞吐量。不管是需要快速访问时间还是快速的传输速度——或混合两者——依赖于正在运行的查询语句的种类。从完成一次磁盘读取所需要的总时间来说，小的随机查找以步骤 1 和 2 为主，而大的顺序读主要是第 3 步。

其他一些因素也可以影响磁盘的选择，哪个重要取决于应用。假设正在为一个在线应用选择磁盘，例如一个受欢迎的新闻网站，有大量小的磁盘随机读取。可能需要考虑下列因素：

存储容量

对在线应用来说容量很少成为问题，因为现在的磁盘通常足够大了。如果不够，用 RAID 把小磁盘组合起来是标准做法[注6]。

传输速度

现代磁盘通常数据传输速度非常快，正如我们前面看到的。究竟多快主要取决于主轴转速和数据存储在磁盘表面上的密度，再加上主机系统的接口的限制（许多现代磁盘读取数据的速度比接口可以传输的快）。无论如何，传输速度通常不是在线应用的限制因素，因为它们一般会做很多小的随机查找。

访问时间

对随机查找的速度而言，这通常是个主要因素，所以应该寻找更快的访问时间的磁盘。

主轴转速

现在常见的转速是 7 200RPM、10 000RPM，以及 15 000RPM。转速不管对随机查找还是顺序扫描都有很大影响。

物理尺寸

所有其他条件都相同的情况下，磁盘的物理尺寸也会带来差别：越小的磁盘，移动

注 6：有趣的是，有些人故意买更大容量的磁盘，然后只使用 20% ~ 30% 的容量。这增加了数据局部性和减少寻道时间，有时可以证明值得它们高的价格。

读取磁头需要的时间就越短。服务器级的 2.5 英寸磁盘性能往往比它们的更大的盘更快。它们还可以节省电力，并且通常可以融入机箱中。

和 CPU 一样，MySQL 如何扩展到多个磁盘上取决于存储引擎和工作负载。InnoDB 能很好地扩展到多个硬盘驱动器。然而，MyISAM 的表锁限制其写的可扩展性，因此写繁重的工作加在 MyISAM 上，可能无法从多个驱动器中收益。虽然操作系统的文件系统缓冲和后台并发写入会有点帮助，但 MyISAM 相对于 InnoDB 在写可扩展性上有更多的限制。

和 CPU 一样，更多的磁盘也并不总是更好。有些应用要求低延迟需要的是更快的驱动器， 400
而不是更多的驱动器。例如，复制通常在更快的驱动器上表现更好，因为备库的更新是单线程的。[注 7]

9.4 固态存储

固态（闪存）存储器实际上是有 30 年历史的技术，但是它作为新一代驱动器而成为热门则是最近几年的事。固态存储现在越来越便宜，并且也更成熟了，它正在被广泛使用，并且可能会在不久的将来在多种用途上代替传统磁盘。

固态存储设备采用非易失性闪存芯片而不是磁性盘片组成。它们也被称为 NVRAM，或非易失性随机存取存储器。固态存储设备没有移动部件，这使得它们表现得跟硬盘驱动器有很大的不同。我们将详细探讨其差异。

目前 MySQL 用户感兴趣的技术可分为两大类：SSD（固态硬盘）和 PCIe 卡。SSD 通过实现 SATA（串行高级技术附件）接口来模拟标准硬盘，所以可以替代硬盘驱动器，直接插入服务器机箱中的现有插槽。PCIe 卡使用特殊的操作系统驱动程序，把存储设备作为一个块设备输出。PCIe 和 SSD 设备有时可以简单地都认为是 SSD。

下面是闪存性能的快速小结。高质量闪存设备具备：

- 相比硬盘有更好的随机读写性能。闪存设备通常读明显比写要快。
- 相比硬盘有更好的顺序读写性能。但是相比而言不如随机 I/O 的改善那么大，因为硬盘随机 I/O 比顺序 I/O 要慢得多。入门级固态硬盘的顺序读取实际上还可能比传统硬盘慢。
- 相比硬盘能更好地支持并发。闪存设备可以支持更多的并发操作，事实上，只有大量的并发请求才能真正实现最大吞吐量。

注 7： 5.6 也可以按库做多线程复制。——译者注

最重要的事情是提升随机 I/O 和并发性。闪存记忆体可以在高并发下提供很好的随机 I/O 性能，这正是范式化的数据库所需要的。设计非范式化的 Schema 最常见的原因之一是为了避免随机 I/O，并且使得查询可能转化为顺序 I/O。

因此，我们相信固态存储未来将从根本上改变 RDBMS 技术。当前这一代的 RDBMS 技
401 术几十年来都是为机械磁盘做优化的。同样成熟和深入的研究工作在固态存储上还没有真正出现[注8]。

9.4.1 闪存概述

硬盘驱动器使用旋转盘片和可移动磁头，其物理结构决定了磁盘固有的局限性和特征。对固态存储也是一样，它是构建在闪存之上的。不要以为固态存储很简单，实际上比硬盘驱动器在某些方面更复杂。闪存的限制实际上是相当严重的，并且难以克服，所以典型的固态设备都有错综复杂的架构、缓存，以及独有的“法宝”。

闪存的最重要的特征是可以迅速完成多次小单位读取，但是写入更有挑战性。闪存不能在没有做擦除操作前改写一个单元（Cell）[注9]，并且一次必须擦除一个大块——例如，512 KB。擦除周期是缓慢的，并且最终会磨损整个块。一个块可以容忍的擦除周期次数取决于所使用的底层技术，有关这些内容我们稍后再讲。

写入的限制是固态存储复杂的原因。这也是为什么一些设备供应商在设备的稳定、性能的一致性等方面和其他供应商有区别的原因。“魔法”全部都在其专有的固件、驱动程序，以及其他零零碎碎的东西里，这些东西使得固态设备良好运转。为了使写入表现良好，并避免闪存块过早损耗完寿命，设备必须能够搬迁页面并执行垃圾收集和所谓的磨损均衡。写放大用于描述数据从一个地方移动到另一个地方的额外写操作，多次写数据和元数据导致局部块经常写。如果你有兴趣，维基百科中的写放大的文章，是个学习的好地方，可以从其中了解更多关于闪存的知识。

垃圾收集对理解闪存很重要。为了保持一些块是干净的并且可以被写入，设备需要回收脏块。这需要设备上有一些空闲空间。无论是设备内部有一些看不到的预留空间，或者通过不写那么多数据来预留需要的空间——不同的设备可能有所不同。无论哪种方式，设备填满了，垃圾收集就必须更加努力地工作，以保持一些块是干净的，所以写放大的倍数就增加了。

因此，许多设备在被填满后会开始变慢。到底会慢多少，不同的制造商和型号之间有所不同，依赖于设备的架构。有些设备为高性能而设计，即使写得非常满，依然可以保持

注 8： 有些公司声称，他们抛弃过去主轴（机械）的羁绊，从一个干净的石板开始。温和的怀疑是有道理的；解决 RDBMS 的挑战是不容易的。

注 9： 这是一种简化，但细节在这里并不重要。如果你喜欢，可以阅读维基百科上的更多信息。

高性能。但是，通常一个 100GB 的文件在 160GB 和 320GB 的 SSD 上表现完全不同。速度下降是由于没有空闲块时必须等待擦写完成所造成的。写到一个空闲块只需要花费数百微秒，但是擦写慢得多——通常需要几个毫秒。 402

9.4.2 闪存技术

有两种主要的闪存设备类型，当考虑购买闪存存储时，理解两者之间的不同是很重要的。这两种类型分别是单层单元（SLC）和多层单元（MLC）。

SLC 的每个单元存储数据的一个比特：可以是 0 或 1。SLC 相对更昂贵，但非常快，并且擦写寿命高达 100 000 个写周期，具体值取决于供应商和型号。这听起来好像不多，但在现实中一个好的 SLC 设备应该持续使用大约 20 年左右，甚至比卡上安装的控制器更耐用和可靠。缺点则是存储密度相对较低，所以不能在每个设备上得到那么多空间。

MLC 每个单元存储 2 个比特、3 个比特的设备也正在进入市场。这使得通过 MLC 设备获得更高的存储密度（更大的容量）成为可能。成本更低了，但是速度和耐擦写性也下降了。一个不错的 MLC 设备可能被定为 10 000 个写循环周期。

可以在大众市场上购买到这两种类型的闪存设备，它们之间的竞争有助于闪存的发展。目前，SLC 仍持有“企业”级服务器的存储解决方案的声誉，通常被视为消费级的 MLC 设备，一般使用在笔记本电脑和数码相机等地方。然而，这种情况正在改变，出现了一种新兴的所谓企业级 MLC（eMLC）存储。

MLC 技术的发展是很有意思的，如果正在考虑购买闪存存储，这个发展方向值得密切关注。MLC 非常复杂，包含很多有助于设备质量和性能的重要因素。任何给定的芯片仅靠自身是不能持久化的，因为有着相对较短的信号保持周期，以及较高的错误率必须纠正。随着市场转移到更小、密度更高的芯片，其中的芯片单元可以存储 3 比特，单个芯片变得更不可靠以及更容易出错。

然而，这并不是一个不可逾越的工程问题。厂商正在制造一些有越来越多隐藏容量的设备，因此有足够的内部冗余。尽管闪存厂商非常注意保护自己的商业秘密，还是有传言称，某些设备可能有比它标称大小多出高达两倍的存储空间。使 MLC 芯片更耐用的另一种方法是通过固件逻辑。平衡磨损和重映射的算法是非常重要的。

寿命的长短取决于真实的容量，固件逻辑等——所以最终是因供应商而异的。我们听说过在几个星期里密集使用导致设备报废的报告！

因此，MLC 设备最关键的环节是内置的算法和智能。制造一个好的 MLC 设备比制造一个 SLC 设备难得多，但也是可能的。随着工程学的伟大进步，以及容量和密度的增加， 403

一些最好的供应商提供的设备，是值得用 eMLC 这个标签的。这个领域随着时间的推移进步得很快，本书对 MLC 与 SLC 的意见可能很快会变得过时。

> **设备的寿命还剩多久?**
>
> Virident 担保其 FlashMax 1.4 TB MLC 设备可以持续写入 15 PB 数据，但这是在闪存级别的数据，用户可见的写入是会放大的。我们跑了一个小实验来发现特定的工作负载下的写入放大因子。
>
> 我们创建了一个 500GB 的数据集，然后在上面运行 *tpcc-mysql* 基准测试，跑了一个小时。在这个小时里，`/proc/diskstats` 报告了 984GB 的写入，然后 Virident 配置工具显示在闪存层有 1 125GB 的写入，因此写入放大因子是 1.14。记住，如果设备上消耗了更多空间，这个值会更高，并且这个值的浮动还基于写入方式是顺序还是随机。
>
> 在这样的比率下，如果不间断地跑一年半的基准测试，就可以用完设备的寿命。当然，真实的工作负载很少是写密集型的，所以这个卡在实际使用中应该可以持续工作很多年。这个观点不是说该设备将很快磨损——它是说，写放大系数是很难预测的，需要检查设备，根据工作量来查看它的行为。
>
> 容量对寿命的影响也很大，正如我们已经提到的。更大容量的设备会使得寿命显著增长，这是为什么 MLC 越来越流行——最近我们看到足够大的容量可以延长寿命是有理由的。

9.4.3 闪存的基准测试

对闪存设备进行基准测试是复杂并且困难的。有很多情况会导致测试错误，需要了解特定设备的知识，并且需要有极大的耐心和关注，才能正确地操作。

闪存设备有一个三阶段模式，我们称为 A-B-C 性能特性。它们开始阶段运行非常快（阶段 A），然后垃圾回收器开始工作，这将导致在一段时间内，设备处于过渡到稳定状态（阶段 B）的阶段，最后设备进入一个稳定状态（状态 C）。所有我们测试过的设备都有这个特点。

当然，我们感兴趣的是阶段 C 的性能，所以基准测试只需要测量这个部分的运行过程。这意味着基准测试要做的不仅仅是基准测试：还需要先进行一下预热，然后才能进行基

准测试。但是，定义预热的终点和基准测试的起点会非常棘手。

设备、文件系统，以及操作系统通过不同方式提供 TRIM 命令的支持，这个命令标记空间准备重用。有时当删除所有文件时设备会被 TRIM。如果在基准测试运行的情况下发生，设备将重置到阶段 A，然后必须重新执行 A 和 B 之间的运行阶段。另一个因素是设备被填充得很满或者不满时，不同的性能表现。一个可重复的基准测试必须覆盖到所有这些因素。 404

通过上述分析，可知基准测试的复杂性，所以就算厂商如实地报告测试结果，但对于外行来说，厂商的基准测试和规格说明书依然可能有很多“坑”。

通常可以从供应商那得到四个数字。这里有一个设备规格的例子 ：

1. 设备读取性能最高达 520 MB/s。
2. 设备写入性能最高达 480 MB/s。
3. 设备持续写入速度可以稳定在 420 MB/s。
4. 设备每秒可以执行 70 000 个 4 KB 的写操作。

如果再次复核这些数字，你会发现峰值 4KB 写入达到 70 000 个 IOPS（每秒输入 / 输出操作)，这么算每秒写入大约只有 274 MB/s，这比第二点和第三点中说明的高峰写入带宽少了很多。这是因为达到峰值写入带宽时是用更大的块，例如 64 KB 或 128 KB。用更小的块大小来达到峰值 IOPS。

大部分应用不会写这么大的块。InnoDB 写操作通常是 16KB 或 512 字节的块组合到一起写回。因此，设备应该只有 274MB/s 的写出带宽——这是阶段 A 的情况，在垃圾回收器开启和设备达到长期稳定的性能等级前！

在我们的博客中，可以找到目前的 MySQL 基准测试，以及在固态硬盘上裸设备文件 I/O 的工作负载 ：*http://www.ssdperformanceblog.com* 和 *http://www.mysqlperformanceblog.com*。

9.4.4 固态硬盘驱动器（SSD）

SSD 模拟 SATA 硬盘驱动器。这是一个兼容性功能 ：替换 SATA 硬盘不需要任何特殊的驱动程序或接口。

英特尔的 X-25E 驱动器可能是我们今天看到在服务器中最常见的固态硬盘，但也有很多其他选择。X-25E 是为“企业”级消费市场开发的，但也有用 MLC 存储的 X-25M，这是为笔记本电脑用户等大众市场准备的。此外，英特尔还销售 320 系列，也有很多人正

在使用。再次，这仅仅是一个供应商——还有很多，在这本书去印刷的时候，我们所写的关于 SSD 的一些东西可能已经过时。

405 关于 SSD 的好处是，它们有大量的品牌和型号相对是比较便宜的，同时它们比硬盘快了很多。最大的缺点是，它们并不总是像硬盘一样可靠，这取决于品牌和型号。直到最近，大多数设备都没有板载电池，但大多数设备上有一个写缓存来缓冲写入。写入缓存在没有电池备份的情况下并不能持久化，但是在快速增长的写负载下，它不能关闭，否则闪存存储无法承受。所以，如果禁用了驱动器的高速缓存以获得真正持久化的存储，将会更快地耗完设备寿命，在某些情况下，这将导致保修失效。

有些厂家完全不急于告诉购买他们固态硬盘的客户关于 SSD 的特点，并且他们对设备的内部架构等细节守口如瓶。是否有电池或电容保护写缓存的数据安全，在电源故障的情况下，通常是一个悬而未决的问题。在某些情况下，驱动器会接受禁用缓存的命令，但忽略了它。所以，除非做过崩溃试验，否则真的有可能不知道驱动器是否是持久化的，我们对一些驱动器进行了崩溃测试，发现了不同的结果。如今，一些驱动器有电容器保护缓存，使其可以持久化，但一般来说，如果驱动器不是自夸有一个电池或电容，那么它就没有。这意味着在断电的情况下不是持久化的，所以可能出现数据已经损坏却还不知情的情况。SSD 是否配置电容或电池是我们必须关注的特性。

通常，使用 SSD 都是值得的。但底层技术的挑战是不容易解决的。很多厂家做出的驱动器在高负载下很快就崩溃了，或不提供持续一致的性能。一些低端的制造商有一个习惯，每次发布新一代驱动器，就声称他们已经解决了老一代的所有问题。这往往是不真实的，当然，如果关心可靠性和持续的高性能，“企业级”的设备通常值得它的价钱。

用 SSD 做 RAID

我们建议对 SATA SSD 盘使用 RAID（Redundant Array of Inexpensive Disks，磁盘冗余阵列）。单一驱动器的数据安全是无法让人信服的。

许多旧的 RAID 控制器并不支持 SSD。因为它们假设管理的是机械硬盘，包括写缓冲和写排序这些特性都是为机械硬盘而设计的。这不但纯属无效工作，也会增加响应时间，因为 SSD 暴露的逻辑位置会被映射到底层闪存记忆体中的任意位置。现在这种情况好一点。有些 RAID 控制器的型号末尾有一个字母，表明它们是为 SSD 做了准备的。例如，Adaptec 控制器用 Z 标识。

然而，即使支持闪存的控制器，也不一定真的就对闪存支持很好。例如，Vadim 对 Adaptec 5805Z 控制器进行了基准测试，他用了多种驱动器做 RAID 10，16 个并发操作

500 GB 的文件。结果是很糟糕的：95% 的随机写延迟在两位数的毫秒，在最坏的情况下，超过一秒钟[注 10]。（期望的应该是亚毫秒级写入。） 406

这种特定的比较，是一家客户为了看到 Micron SSD 是否会比 64GB 的 Intel SSD 更好而做的，该比较是基于相同的配置的。当为英特尔驱动器进行基准测试时，我们发现了相同的性能特征。因此，我们尝试了一些其他驱动器的配置，不管有没有 SAS 扩展器，看看会发生什么。表 9-1 显示了这个结果。

表9-1：在Adaptec RAID 控制器上用SSD进行的基准测试

驱动器数量	厂家	大小	SAS扩展器	随机读	随机写
34	Intel	64 GB	Yes	310 MB/s	130 MB/s
14	Intel	64 GB	Yes	305 MB/s	145 MB/s
24	Micron	50 GB	No	350 MB/s	120 MB/s
34	Intel	50 GB	No	350 MB/s	180 MB/s

这些结果都没有达到我们对这么多驱动器的期望。在一般情况下，RAID 控制器的性能表现，只能满足对 6 ~ 8 个驱动器的期望，而不是几十个。原因很简单，RAID 控制器达到了瓶颈。这个故事的重点是，在对硬件投入巨资前，应该先仔细进行基准测试——结果可能与期望的有相当大的区别。

9.4.5 PCIe 存储设备

相对于 SATA SSD，PCIe 设备没有尝试模拟硬盘驱动器。这种设计是好事：服务器和硬盘驱动器之间的接口不能完全发挥闪存的性能。SAS/SATA 互联带宽比 PCIe 要低，所以 PCIe 对高性能需求是更好的选择。PCIe 设备延迟也低得多，因为它们在物理上更靠近 CPU。

没有什么比得上从 PCIe 设备上获得的性能。缺点就是它们太贵了。

所有我们熟悉的型号都需要一个特殊的驱动程序来创建块设备，让操作系统把它认成一个硬盘驱动器。这些驱动程序使用着混合磨损均衡和其他逻辑的策略；有些使用主机系统的 CPU 和内存，有些使用板载的逻辑控制器和 RAM（内存）。在许多场景下，主机系统有丰富的 CPU 和 RAM 资源，所以相对于购买一个自身有这些资源的卡，利用主机上的资源实际上是更划算的策略。

我们不建议对 PCIe 设备建 RAID。它们用 RAID 就太昂贵了，并且大部分设备无论以何种方式，都有它们自己板载的 RAID。我们并不真的知道控制器坏了以后会怎么样，但是厂商说他们的控制器通常跟网卡或者 RAID 控制器一样好，看起来确实是这样。换 407

注 10： 但这不是全部。我们在基准测试后检查了驱动器，并且发现两块 SSD 坏盘，有一块不一致。

句话说，这些设备的平均无故障时间间隔（MTBF）接近于主板，所以对这些设备使用RAID只是增加了大量成本而没有多少好处。

有许多家供应商在生产PCIe闪存卡。对MySQL用户来说最著名的厂商是Fusion-io和Virident，但是像Texas Memory Systems、STEC和OCZ这样的厂商也有用户。SLC和MLC都有相应的PCIe卡产品。

9.4.6 其他类型的固态存储

除了SSD和PCIe设备，也有其他公司的产品可以选择，例如Violin Memory、SandForce和Texas Memory Systems。这些公司提供有几十TB存储容量，本质上是闪存SAN的大箱子。它们主要用于大型数据中心存储的整合。虽然价格非常昂贵，但是性能也非常高。我们知道一些使用的例子，并在某些场景下测量过性能。这些设备能够提供相当好的延迟，除了网络往返时间——例如，通过NFS有小于4毫秒的延迟。

然而这些都不适合一般的MySQL市场。它们的目标更针对其他数据库，如Oracle，可以用来做共享存储集群。一般情况下，MySQL在如此大规模的场景下，不能有效利用如此强大的存储优势，因为在数十个TB的数据下MySQL很难良好地工作——MySQL对这样一个庞大的数据库的回答是，拆分、横向扩展和无共享（Shared-nothing）架构。

虽然专业化的解决方案可能能够利用这些大型存储设备——例如Infobright可能成为候选人。ScaleDB可以部署在共享存储（Shared-storage）架构，但我们还没有看到它在生产环境应用，所以我们不知道其工作得如何。

9.4.7 什么时候应该使用闪存

固态存储最适合使用在任何有着大量随机I/O工作负载的场景下。随机I/O通常是由于数据大于服务器的内存导致的。用标准的硬盘驱动器，受限于转速和寻道延迟，无法提供很高的IOPS。闪存设备可以大大缓解这种问题。

当然，有时可以简单地购买更多内存，这样随机工作负载就可以转移到内存，I/O就不存在了。但是当无法购买足够的内存时，闪存也可以提供帮助。另一个不能总是用内存解决的问题是，高吞吐的写入负载。增加内存只能帮助减少写入负载到磁盘，因为更多的内存能创造更多的机会来缓冲、合并写。这允许把随机写转换为更加顺序的I/O。

408 然而，这并不能无限地工作下去，一些事务或插入繁忙的工作负载不能从这种方法中获益。闪存存储在这种情况下却也有帮助。

单线程工作负载是另一个闪存的潜在应用场景。当工作负载是单线程的时候，它是对延

迟非常敏感的，固态存储更低的延迟可以带来很大的区别。相反，多线程工作负载通常可以简单地加大并行化程度以获得更高的吞吐量。MySQL 复制是单线程工作的典型例子，它可以从低延迟中获得很多收益。在备库跟不上主库时，使用闪存存储往往可以显著提高其性能。

闪存也可以为服务器整合提供巨大的帮助，尤其是 PCIe 方式的。我们已经看到了机会，把很多实例整合到一台物理服务器——有时高达 10 或 15 倍的整合都是可能的。更多关于这个话题的信息，请参见第 11 章。

然而闪存也可能不一定是你要的答案。一个很好的例子是，像 InnoDB 日志文件这样的顺序写的工作负载，闪存不能提供多少成本与性能优势，因为在这种情况下，闪存连续写方面不比标准硬盘快多少。这样的工作负载也是高吞吐的，会更快耗尽闪存的寿命。在标准硬盘上存放日志文件通常是一个更好的主意，用具有电池保护写缓存的 RAID 控制器。

有时答案在于内存/磁盘的比例，而不只是磁盘。如果可以买足够的内存来缓存工作负载，就会发现这更便宜，并且比购买闪存存储设备更有效。

9.4.8 使用 Flashcache

虽然有很多因素需要在闪存、硬盘和 RAM 之间权衡，在存储层次结构中，这些设备没有被当作一个整体处理。有时可以使用磁盘和内存技术的结合，这就是 Flashcache。

Flashcache 是这种技术的一个实现，可以在许多系统上发现类似的使用，例如 Oracle 数据库、ZFS 文件系统，甚至许多现代的硬盘驱动器和 RAID 控制器。下面讨论的许多东西应用广泛，但我们将只专注于 Flashcache，因为它和厂商、文件系统无关。

Flashcache 是一个 Linux 内核模块，使用 Linux 的设备映射器（Device Mapper）。它在内存和磁盘之间创建了一个中间层。这是 Facebook 开源和使用的技术之一，可以帮助其优化数据库负载。

Flashcache 创建了一个块设备，并且可以被分区，也可以像其他块设备一样创建文件系统，特点是这个块设备是由闪存和磁盘共同支撑的。闪存设备用作读取和写入的智能高速缓存。

虚拟块设备远比闪存设备要大，但是没关系，因为数据最终都存储在磁盘上。闪存设备只是去缓冲写入和缓存读取，有效弥补了服务器内存容量的不足[注11]。 409

注 11： 意思就是内存放不下要缓存的数据时，换出到 Flashcache 上，Flashache 的闪存设备可以帮助继续缓存，而不会立刻落到磁盘。——译者注

这种性能有多好呢？ Flashcache 似乎有相对较高的内核开销。（设备映射并不总是像看起来那么有效，但我们还没深入调查找出原因。）但是，尽管 Flashcache 理论上可能更高效，但最终的性能表现并不如底层的闪存存储那么好，不过它仍然比磁盘快很多，所以还是值得考虑的方案。

我们用包含数百个基准测试的一系列测试来评估 Flashcache 的性能，但是我们发现在人工模拟的工作负载下，测出有意义的数据是非常困难的。于是我们得出结论，虽然并不清楚 Flashcache 通常对写负载有多大好处，但是对读肯定是有帮助的。于是它适合这样的情况使用：有大量的读 I/O，并且工作集比内存大得多。

除了实验室测试，我们有一些生产环境中应用 Flashcache 的经验。想到的一个例子是，有个 4TB 的数据库，这个数据库遇到了很大的复制延迟。我们给系统加了半个 TB 的 Virident PCIe 卡作为存储。然后安装了 Flashcache，并且把 PCIe 卡作为绑定设备的闪存部分，复制速度就翻了一倍。

当闪存卡用得很满时使用 Flashcache 是最经济的，因此选择一张写得很满时其性能不会降低多少的卡非常重要。这就是为什么我们选择 Virident 卡。

Flashcache 就是一个缓存系统，所以就像任何其他缓存一样，它也有预热问题。虽然预热时间可能会非常长。例如，在我们刚才提到的情况下，Flashcache 需要一个星期的预热，才能真正对性能产生帮助。

应该使用 Flashcache 吗？根据具体情况可能会有所不同，所以我们认为在这一点上，如果你觉得不确定，最好得到专家的意见。理解 Flashcache 的机制和它们如何影响你的数据库工作集大小是很复杂的，在数据库下层（至少）有三层存储：

- 首先，是 InnoDB 缓冲池，它的大小跟工作集大小一起可以决定缓存的命中率。缓存命中是非常快的，响应时间非常均匀。
- 在缓冲池中没有命中，就会到 Flashcache 设备上去取，这就会产生分布比较复杂的响应时间。Flashcache 的缓存命中率由工作集大小和闪存设备大小决定。从闪存上命中比在磁盘上查找要快得多。
- Flashcache 设备缓存也没有命中，那就得到磁盘上找，这也会看到分布相当均匀的比较慢的响应时间。

410 有可能还有更多层次：例如，SAN 或 RAID 控制器的缓存。

这有一个思维实验，说明这些层是如何交互的。很显然，从 Flashcache 设备访问的响应时间不会像直接访问闪存设备那么稳定和高速。但是想象一下，假设有 1TB 的数据，其中 100 GB 在很长一段时间会承受 99% 的 I/O 操作。也就是说，大部分时候 99% 的工作

集只有 100 GB。

现在，假设有以下的存储设备：一个很大的 RAID 卷，可以执行 1 000 IOPS，以及一个可以达到 100 000 IOPS 的更小的闪存设备。闪存设备不足以存放所有的数据——假设只有 128 GB——因此单独使用闪存不是一种可能的选择。如果用闪存设备做 Flashcache，就可以期望缓存命中远远快于磁盘检索，但 Flashcache 整体比单独使用闪存设备要慢。我们坚持用数字说话，如果 90% 的请求落到 Flashcache 设备，相当于达到 50 000 IOPS。

这个思维实验的结果是什么呢？有两个要点：

1. 系统使用 Flashcache 比不使用的性能要好很多，因为大多数在缓冲池未命中的页面访问都被缓存在闪存卡上，相对于磁盘可以提供快得多的访问速度。（99% 的工作集可以完全放在闪存卡上。）
2. Flashcache 设备上有 90% 的命中率意味着有 10% 没有命中。因为底层的磁盘只能提供 1 000 IOPS，因此整个 Flashcache 设备可以支持 10 000 的 IOPS。为了明白为什么是这样的，想象一下如果我们要求不止于此会发生什么：10% 的 I/O 操作在缓存中没有命中而落到了 RAID 卷上，则肯定要求 RAID 卷提供超过 1 000 IOPS，很显然是没法处理的。因此，即使 Flashcache 比闪存卡慢，系统作为一个整体仍然受限于 RAID 卷，不止是闪存卡或 Flashcache。

归根到底，Flashcache 是否合适是一个复杂的决定，涉及的因素很多。一般情况下，它似乎最适合以读为主的 I/O 密集型负载，并且工作集太大，用内存优化并不经济的情况。

9.4.9 优化固态存储上的 MySQL

如果在闪存上运行 MySQL，有一些配置参数可以提供更好的性能。InnoDB 的默认配置从实践来看是为硬盘驱动器定制的，而不是为固态硬盘定制的。不是所有版本的 InnoDB 都提供同样等级的可配置性。尤其是很多为提升闪存性能设计的参数首先出现在 Percona Server 中，尽管这些参数很多已经在 Oracle 版本的 InnoDB 中实现，或者计划在未来的版本中实现。

改进包括：

增加 InnoDB 的 I/O 容量 411

闪存比机械硬盘支持更高的并发量，所以可以增加读写 I/O 线程数到 10 或 15 来获得更好的结果。也可以在 2 000 ~ 20 000 范围内调整 `innodb_io_capacity` 选项，这要看设备实际上能支撑多大的 IOPS。尤其是对 Oracle 官方的 InnoDB 这个很有必要，内部有更多算法依赖于这个设置。

让 InnoDB 日志文件更大

即使最近版本的 InnoDB 中改进了崩溃恢复算法，也不应该把磁盘上的日志文件调得太大，因为崩溃恢复时需要随机 I/O 访问，会导致恢复需要很长一段时间。闪存存储让这个过程快很多，所以可以设置更大的 InnoDB 日志文件，以帮助提升和稳定性能。对于 Oracle 官方的 InnoDB，这个设置尤其重要，它维持一个持续的脏页刷新比例有点麻烦，除非有相当大的日志文件——4GB 或者更大的日志文件，在写的时候对服务器来说是个不错的选择。Percona Server 和 MySQL 5.6 支持大于 4GB 的日志文件。

把一些文件从闪存转移到 RAID

除了把 InnoDB 日志文件设置得更大，把日志文件从数据文件中拿出来，单独放在一个带有电池保护写缓存的 RAID 组上而不是固态设备上，也是个好主意。这么做有几个原因。一个原因是日志文件的 I/O 类型，在闪存设备上不比在这样一个 RAID 组上要快。InnoDB 写日志是以 512 字节为单位的顺序 I/O 写下去，并且除了崩溃恢复会顺序读取，其他时候绝不会去读。这样的 I/O 操作类型用闪存设备是很浪费的。并且把小的写入操作从闪存转移到 RAID 卷也是个好主意，因为很小的写入会增加闪存设备的写放大因子，会影响一些设备的使用寿命。大小写操作混合到一起也会引起某些设备延时的增加。

基于相同的原因，有时把二进制日志文件转移到 RAID 卷也会有好处。并且你可能会认为 *ibdata1* 文件也适合放在 RAID 卷上，因为 *ibdata1* 文件包含双写缓冲（Doublewrite Buffer）和插入缓冲（Insert Buffer），尤其是双写缓冲会进行很多重复写入。在 Percona Server 中，可以把双写缓冲从 *ibdata1* 文件中拿出来，单独存放到一个文件，然后把这个文件放在 RAID 卷上。

还有另一个选择：可以利用 Percona Server 的特性，使用 4KB 的块写事务日志，而不是 512 字节。因为这会匹配大部分闪存本身的块大小，所以可以获得更好的效果。所有的上述建议是对特定硬件而言的，实际操作的时候可能会有所不同，所以在大规模改动存储布局之前要确保已经理解相关的因素——并辅以适当的测试。

412 *禁用预读*

预读通过通知和预测读取模式来优化设备的访问，一旦认为某些数据在未来需要被访问到，就会从设备上读取这些数据。实际上在 InnoDB 中有两种类型的预读，我们发现在多种情况下的性能问题，其实都是预读以及它的内部工作方式造成的。在许多情况下开销比收益大，尤其是在闪存存储，但我们没有确凿的证据或指导，禁用预读究竟可以提高多少性能。

在 MySQL 5.1 的 InnoDB Plugin 中，MySQL 禁用了所谓的“随机预读”，然后在 MySQL 5.5 又重新启用了它，可以在配置文件用一个参数配置。Percona Server 能让你在旧版本里也一样可以配置为 random（随机）或 linear read-ahead（线性预读）。

配置 InnoDB 刷新算法

这决定 InnoDB 什么时候、刷新多少、刷新哪些页面，这是个非常复杂的主题，这里我们没有足够的篇幅来讨论这些具体的细节。这也是个研究比较活跃的主题，并且实际上在不同版本的 InnoDB 和 MySQL 中有多种有效的算法。

标准 InnoDB 算法没有为闪存存储提供多少可配置性，但是如果用的是 Percona XtraDB（包含在 Percona Server 和 MariaDB 中），我们建议设置 `innodb_adaptive_checkpoint` 选项为 `keep_average`，不要用默认值 `estimate`。这可以确保更持续的性能，并且避免服务器抖动，因为 `estimate` 算法会在闪存存储上引起抖动。我们专门为闪存存储开发了 `keep_average`，因为我们意识到对于闪存设备，把希望操作的大量 I/O 推到设备上，并不会引起瓶颈或发生抖动。

另外，建议为闪存设备设置 `innodb_flush_neighbor_pages = 0`。这样可以避免 InnoDB 尝试查找相邻的脏页一起刷写。这个算法可能会导致更大块的写、更高的延迟，以及内部竞争。在闪存存储上这完全没必要，也不会有什么收益，因为相邻的页面单独刷新不会冲击性能。

禁用双写缓冲的可能

相对于把双写缓存转移到闪存设备，可以考虑直接关闭它。有些厂商声称他们的设备支持 16KB 的原子写入，使得双写缓冲成为多余的。如果需要确保整个存储系统被配置得可以支持 16KB 的原子写入，通常需要 `O_DIRECT` 和 XFS 文件系统。

没有确凿的证据表明原子操作的说法是真实的，但由于闪存存储的工作方式，我们相信写数据文件发生页面写一部分的情况是大大减少的，并且这个收益在闪存设备上比在传统磁盘上要高得多，禁用双写缓冲在闪存存储上可以提高 MySQL 整体性能差不多 50%，尽管我们不知道这是不是 100% 安全的，但是你可以考虑下这么做。 413

限制插入缓冲大小

插入缓冲（在新版 InnoDB 中称为变更缓冲（Change Buffer））设计来用于减少当更新行时不在内存中的非唯一索引引起的随机 I/O。在硬盘驱动器上，减少随机 I/O 可以带来巨大的性能提升。对某些类型的工作负载，当工作集比内存大很多时，差异可能达到近两个数量级。插入缓冲在这类场景下就很有用。

然而，对闪存就没有必要了。闪存上随机 I/O 非常快，所以即使完全禁用插入缓冲，也不会带来太大影响，尽管如此，可能你也不想完全禁用插入缓存。所以最好还是启用，因为 I/O 只是修改不在内存中的索引页面的开销的一部分。对闪存设备而言，最重要的配置是控制最大允许的插入缓冲大小，可以限制为一个相对比较小的值，而不是让它无限制地增长，这可以避免消耗设备上的大量空间，并避免 *ibdata1* 文件变得非常大的情况。在本书写作的时候，标准 InnoDB 还不能配置插入缓存的容量上限，但是在 Percona XtraDB（Percona Server 和 MariaDB 都包含 XtraDB）里可以。MySQL 5.6 里也会增加一个类似的变量。

除了上述的配置建议，我们还提出或讨论了其他一些闪存优化策略。然而，不是所有的策略都非常容易明白，所以我们只是提到了一部分，最好自己研究在具体情况下的好处。首先是 InnoDB 的页大小。我们发现了不同的结果，所以我们现在还没有一个明确的建议。好消息是，在 Percona Server 中不需要重编译也能配置页面大小，在 MySQL 5.6 中这个功能也可能实现。以前版本的 MySQL 需要重新编译服务器才能使用不同大小的页面，所以大部分情况都是运行在默认的 16KB 页面。当页面大小更容易让更多人进行实验时，我们期待更多非标准页面大小的测试，可能能从中得到很多重要的结论。

另一个提到的优化是 InnoDB 页面校验(Checksum)的替代算法。当存储系统响应很快时，校验值计算可能开始成为 I/O 相关操作中显著影响时间的因素，并且对某些人来说这个计算可能替代 I/O 成为新的瓶颈。我们的基准测试还没有得出可适用于普遍场景的结论，所以每个人的情况可能有所不同。Percona XtraDB 允许修改校验算法，MySQL 5.6 也有了这个功能。

可能已经提醒过了，我们提到的很多功能和优化在标准版本的 InnoDB 中是无效的。我们希望并且相信我们引入 Percona Server 和 XtraDB 中的改进点，最终将会被广大用户接受。与此同时，如果正使用 Oracle 官方 MySQL 分发版本，依然可以对服务器采取措施为闪存进行优化。建议使用 `innodb_file_per_table`，并且把数据文件目录放到闪存设备。然后移动 *ibdata1* 和日志文件，以及其他所有日志文件（二进制日志、复制日志，等等），到 RAID 卷，正如我们之前讨论的。这会把随机 I/O 集中到闪存设备上，然后把大部分顺序写入的压力尽可能转移出闪存，因而可以节省闪存空间并且减少磨损。

414

另外，所有版本的 MySQL 服务器，都应该确认超线程开启了。当使用闪存存储时，这有很大的帮助，因为磁盘通常不再是瓶颈，任务会更多地从 I/O 密集变为 CPU 密集。

9.5 为备库选择硬件

为备库选择硬件与为主库选择硬件很相似，但是也有些不同。如果正计划着建一个备库做容灾，通常需要跟主库差不多的配置。不管备库是不是仅仅作为一个主库的备用库，都应该强大到足以承担主库上发生的所有写入，额外的不利因素是备库只能序列化串行执行。(下一章有更多关于这方面的内容)。

备库硬件主要考虑的是成本：需要在备库硬件上花费跟主库一样多的成本吗？可以把备库配置得不一样以便从备库上获得更多性能吗？如果备库跟主库工作负载不一样，可以从不一样的硬件配置上获得隐含的收益吗？

这一切都取决于备库是否只是备用的，你可能希望主库和备库有相同的硬件和配置，但是，如果只是用复制作为扩展更多读容量的方法，那备库可以有多种不同的捷径。例如，

可能在备库使用不一样的存储引擎，并且有些人使用更便宜的硬件或者用 RAID 0 代替 RAID 5 或 RAID 10。也可以取消一些一致性和持久性的保证让备库做更少的工作。

这些措施在大规模部署的情况下具有很好的成本效益，但是在小规模的情况下，可能只会使事情变得更加复杂。在实践中，似乎大多数人都会选择以下两种策略为备库选择硬件：主备使用相同的硬件，或为主库购买新的硬件，然后让备库使用主库淘汰的老硬件。

在备库很难跟上主库时，使用固态硬盘有很大的意义。很好的随机 I/O 性能有助于缓解单个复制线程的影响。

9.6 RAID 性能优化

415

存储引擎通常把数据和索引都保存在一个大文件中，这意味着用 RAID（Redundant Array of Inexpensive Disks，磁盘冗余阵列）存储大量数据通常是最可行的方法[注12]。RAID 可以帮助做冗余、扩展存储容量、缓存，以及加速。但是从我们看到的一些优化案例来说，RAID 上有多种多样的配置，为需求选择一个合适的配置非常重要。

我们不想覆盖所有的 RAID 等级，或者深入细节来分析不同的 RAID 等级分别如何存储数据。关于这个主题有很多好资料，在一些书籍和在线文档可以找到[注13]。因此，我们专注于怎样配置 RAID 来满足数据库服务器的需求。最重要的 RAID 级别如下：

RAID 0

如果只是简单地评估成本和性能，RAID 0 是成本最低和性能最高的 RAID 配置（但是，如果考虑数据恢复的因素，RAID 0 的代价会非常高）。因为 RAID 0 没有冗余，建议只在不担心数据丢失的时候使用，例如备库或者因某些原因只是“一次性”使用的时候。典型的案例是可以从另一台备库轻易克隆出来的备库服务器。再次说明，RAID 0 没有提供任何冗余，即使 R 在 RAID 中表示冗余。实际上，RAID 0 阵列的损坏概率比单块磁盘要高，而不是更低！

RAID 1

RAID 1 在很多情况下提供很好的读性能，并且在不同的磁盘间冗余数据，所以有很好的冗余性。RAID 1 在读上比 RAID 0 快一些。它非常适合用来存放日志或者类似的工作，因为顺序写很少需要底层有很多磁盘（随机写则相反，可以从并发中受益）。这通常也是只有两块硬盘又需要冗余的低端服务器的选择。

注 12： 分区（看第 7 章）是另一个好办法，因为它通常把文件分成多份，你可以放在不同的磁盘上。但是，相对于分区，RAID 对于很大数据是一个更简单的解决方案。这不需要你手动进行负载平衡或者在负载分布发生变化时进行干预，并且可以提供冗余，而你不能把分区文件放在不同的磁盘。

注 13： 两个很好的 RAID 学习资源是维基百科上的文章（*http://en.wikipedia.org/wiki/RAID*）和 AC&NC 教程 *http://www.acnc.com/04_00.html*。

RAID 0 和 RAID 1 很简单，在软件中很好实现。大部分操作系统可以很简单地用软件创建 RAID 0 和 RAID 1。

416 *RAID 5*

RAID 5 有点吓人，但是对某些应用，这是不可避免的选择，因为价格或者磁盘数量（例如需要的容量用 RAID 1 无法满足）的原因。它通过分布奇偶校验块把数据分散到多个磁盘，这样，如果任何一个盘的数据失效，都可以从奇偶校验块中重建。但如果有两个磁盘失效了，则整个卷的数据无法恢复。就每个存储单元的成本而言，这是最经济的冗余配置，因为整个阵列只额外消耗了一块磁盘的存储空间。

在 RAID 5 上随机写是昂贵的，因为每次写需要在底层磁盘发生两次读和两次写，以计算和存储校验位。如果写操作是顺序的，那么执行起来会好一些，或者有很多物理磁盘也行。另外说一下，随机读和顺序读都能很好地在 RAID 5 下执行注 14。RAID 5 用作存放数据或者日志是一种可接受的选择，或者是以读为主的业务，不需要消耗太多写 I/O 的场景。

RAID 5 最大的性能消耗发生在磁盘失效时，因为数据需要重分布到其他磁盘。这会严重影响性能，如果有很多磁盘会更糟糕。如果在重建数据时还保持服务器在线服务，那就别指望重建的速度或者阵列的性能会好。如果使用 RAID 5，最好有一些机制可以做故障迁移，当有问题的时候让一台机器不再提供服务，另一台接管。不管怎样，对系统做一下故障恢复时的性能测试很有必要，这样就可以知道故障恢复时的性能表现到底如何。如果一块磁盘失效，RAID 组在重建过程中，会导致磁盘性能下降，使用这个存储的服务器整体性能可能会不成比例地被影响到慢两倍到五倍。

RAID 5 的奇偶校验块会带来额外的性能开销，这会限制它的可扩展性，超过 10 块硬盘后 RAID 5 就不能很好地扩展，RAID 缓存也会有些问题。RAID 5 的性能严重依赖于 RAID 控制器的缓存，这可能跟数据库服务器需要的缓存冲突了。我们稍后会讨论缓存。

尽管 RAID 5 有这么多问题，但有个有利因素是它非常受欢迎。因此，RAID 控制器往往针对 RAID 5 做了高度优化，虽然有理论极限，但是智能控制器充分利用高速缓存使得 RAID 5 在某些场景下有时可以达到接近 RAID 10 的性能。实际上这可能反映了 RAID 10 的控制器缺少很好的优化，但不管是什么原因，这就是我们所见到的。

RAID 10

RAID 10 对数据存储是个非常好的选择。它由分片的镜像组成，所以对读和写都有良好的扩展性。相对于 RAID 5，重建起来很简单，速度也很快。另外 RAID 10 还可以在软件层很好地实现。

当失去一块磁盘时，性能下降还是比较明显的，因为条带可能成为瓶颈注 15。性能可

注 14： 因为读取并不需要写校验位。——译者注

注 15： 意思是损失一块盘，读取的时候本来可以从相互镜像的两块盘中同时读，少了一块盘就只能从另一块镜像盘上去读了。——译者注

能下降为 50%，具体要看工作负载。需要注意的一件事是，RAID 控制器对 RAID 417
10 采用了一种“串联镜像”的实现。这不是最理想的实现，由于条带化的缺点是“最经常访问的数据可能仅被放置在一对机械磁盘上，而不是分布很多份，”所以可能会遇到性能不佳的情况。

RAID 50

RAID 50 由条带化的 RAID5 组成，如果有很多盘的话，这可能是 RAID 5 的经济性和 RAID 10 的高性能之间的一个折中。它的主要用处是存放非常庞大的数据集，例如数据仓库或者非常庞大的 OLTP 系统。

表 9-2 是多种 RAID 配置的总结。

表9-2：RAID等级之间的比较

等级	概要	冗余	盘数	读快	写快
RAID 0	便宜，快速，危险	No	N	Yes	Yes
RAID 1	高速读，简单，安全	Yes	2（通常）	Yes	No
RAID 5	安全（速度）成本折中	Yes	N + 1	Yes	依赖于最慢的盘
RAID 10	昂贵，高速，安全	Yes	2N	Yes	Yes
RAID 50	为极大的数据存储服务	Yes	2(N + 1)	Yes	Yes

9.6.1 RAID 的故障转移、恢复和镜像

RAID 配置（除了 RAID 0）都提供了冗余。这很重要，但很容易让人低估磁盘同时发生故障的可能性。千万不要认为 RAID 能提供一个强有力的数据安全性保证[注16]。

RAID 不能消除甚至减少备份的需求。当出现问题的时候，恢复时间要看控制器、RAID 等级、阵列大小、硬盘速度，以及重建阵列时是否需要保持服务器在线。

硬盘在完全相同的时间损坏是有可能的。例如，峰值功率或过热，可以很容易地废掉两个或更多的磁盘。然而，更常见的是，两个密切相关的磁盘[注17]出现故障。许多这样的隐患可能被忽视了。一个常见的情况是，很少被访问的数据，在物理媒介上损坏了。这可能几个月都检测不到，直到尝试读取这份数据，或另一个硬盘也失效了，然后 RAID 控制器尝试使用损坏的数据来重建阵列。越大的硬盘驱动器，越容易发生这种情况。

这就是为什么做 RAID 阵列的监控如此重要。大部分控制器提供了一些软件来报告阵列的状态，并且需要持续跟踪这些状态，因为不这么做可能就会忽略了驱动器失效。你可能丧失恢复数据和发现问题的时机，当第二块硬盘损坏时，已经晚了。因此应该配置一 418
个监控系统来提醒硬盘或者 RAID 卷发生降级或失效了。

注 16： 尤其是 SSD 盘，同时损坏的可能性是比较大的。——译者注
注 17： 例如一份数据的两个镜像就在这两个盘上。——译者注

对阵列积极地进行定期一致性检查，可以减少潜在的损坏风险。某些控制器有后台巡检（Background Patrol Read）功能，当所有驱动器都在线服务时，可以检查媒介是否有损坏并且修复，也可以帮助避免此类问题的发生。在恢复时，非常大型的阵列可能会降低检查速度，所以创建大型阵列时一定要确保制定了相应的计划。

也可以添加一个热备盘，这个盘一般是未使用状态，并且配置为备用状态，有硬盘坏了之后控制器会自动把这块盘恢复为使用状态。如果依赖于每个服务器的可用性[注18]，这是一个好主意。对只有少数硬盘驱动器的服务器，这么做是很昂贵的，因为一个空闲磁盘的成本比例比较高，但如果有多个磁盘，而不设一个热备盘，就是愚蠢的做法。请记住，更多的磁盘驱动器会让发生故障的概率迅速增加。

除了监控硬盘失效，还应该监控 RAID 控制器的电池备份单元以及写缓存策略。如果电池失效，大部分控制器默认设置会禁用写缓存，把缓存策略从 WriteBack 改为 WriteThrough。这可能导致服务器性能下降。很多控制器会通过一个学习过程周期性地对电池充放电，在这个过程中缓存是被禁用的。RAID 控制器管理工具应该可以浏览和配置电池充放电计划，不会让人措手不及。

也许希望把缓存策略设为 WriteThrough 来测试系统，这样就可以知道系统性能的期望值。也许需要计划电池充放电的周期，安排在晚上或者周末，重新配置服务器修改 `innodb_flush_log_at_trx_commit` 和 `sync_binlog` 变量，或者在电池充放电时简单地切换到另一台服务器。

9.6.2 平衡硬件 RAID 和软件 RAID

操作系统、文件系统和操作系统看到的驱动器数量之间的相互作用可以是复杂的。Bug、限制或只是错误配置，都可能会把性能降低到远远低于理论值。

如果有 10 块硬盘，理想中应该能够承受 10 个并行的请求，但有时文件系统、操作系统或 RAID 控制器会把请求序列化。面对这个问题一个可行的办法是尝试不同的 RAID 配置。例如，如果有 10 个磁盘，并且必须使用镜像冗余，性能也要好，可以考虑下面几种配置：

- 配置一个包含五个镜像对（RAID 1）的 RAID 10 卷[注19]。操作系统只会看到一个很大的单独的硬盘卷，RAID 控制器会隐藏底层的 10 块硬盘。
- 419 在 RAID 控制器中配置五个 RAID 1 镜像对，然后让操作系统使用五个卷而不是一个卷。[注20]

注 18：就是每个服务器上的数据都会被业务使用，没有机器作为备用的。——译者注
注 19：就是先做五个两块盘的 RAID 1，然后再把五个镜像对做成 RAID 0，形成 RAID 10。——译者注
注 20：就是做五个两块盘的 RAID 1，然后交给操作系统使用。——译者注

- 在 RAID 控制器中配置五个 RAID 1 镜像对，然后使用软件 RAID 0 把五个卷做成一个逻辑卷，通过部分硬件、部分软件的实现方式，有效地实现了 RAID 10。注21

哪个选项是最好的？这依赖于系统中所有的组件如何相互协作。不同的配置可能获得相同的结果，也可能不同。

我们已经提醒了多种配置可能导致串行化。例如，ext3 文件系统每个 inode 有一个单一的 Mutex，所以当 InnoDB 是配置为 innodb_flush_method=O_DIRECT（常见的配置）时，在文件系统会有 inode 级别的锁定。这使得它不可能对文件进行 I/O 并发操作，因而系统表现会远低于其理论上的能力。

我们见过的另一个案例，请求串行地发送到一个 10 块盘的 RAID10 卷中的每个设备，使用 ReiserFS 文件系统，InnoDB 打开了 innodb_file_per_table 选项。尝试在硬件 RAID 1 的基础上用软件 RAID 0 做成 RAID 10 的方式，获得了五倍多的吞吐，因为存储系统开始表现出五个硬盘同时工作的特性，而不再是一个了。造成这种情况的是一个已经被修复的 Bug，但是这是一个很好的例证，说明这类事情可能发生。

串行化可能发生在任何的软件或硬件堆栈层。如果看到这个问题发生了，可能需要更改文件系统、升级内核、暴露更多的设备给操作系统，或使用不同的软件或硬件 RAID 组合方式。应该检查你的设备的并发性以确保它确实是在做并发 I/O（本章稍后有更多关于这个话题的内容）。

最后，当准备上线一种新服务器时，不要忘了做基准测试！这会帮助你确认能获得所期望的性能。例如，若一个硬盘驱动器每秒可以做 200 个随机读，一个有 8 个硬盘驱动器的 RAID 10 卷应该接近每秒 1 600 个随机读。如果观察到的结果比这个少得多，比如说每秒 500 个随机读，就应该研究下哪里可能有问题了。确保基准测试对 I/O 子系统施加了跟 MySQL 一样的方式的压力——例如，使用 O_DIRECT 标记，并且如果使用没有打开 innodb_file_per_table 选项的 InnoDB，要用一个单一的文件测试 I/O 性能。通常可以使用 *sysbench* 来验证新的硬件设置都是正确的。

9.6.3 RAID 配置和缓存

配置 RAID 控制器通常有几种方法，一是可以在机器启动时进入自带的设置工具，或从命令行中运行。虽然大多数控制器提供了很多选项，但其中有两个是我们特别关注的，一是条带化阵列的块大小（*Chunk Size*），还有就是控制器板载缓存（也称为 RAID 缓存，我们使用术语）。

注 21：有些 RAID 卡不支持直接做 RAID 10，只能做成几组 RAID 1，然后由操作系统 LVM 再做 RAID 0，最终形成 RAID 10。——译者注

420 RAID 条带块大小

最佳条带块大小和具体工作负载以及硬件息息相关。从理论上讲，对随机 I/O 来说更大的块更好，因为这意味着更多的读取可以从一个单一的驱动器上满足。

为什么会是这样？在工作负载中找出一个典型的随机 I/O 操作，如果条带的块大小足够大，至少大到数据不用跨越块的边界，就只有单个硬盘需要参与读取。但是，如果块大小比要读取的数据量小，就没有办法避免多个硬盘参与读取。

这只是理论上的观点。在实践中，许多 RAID 控制器在大条带下工作得不好。例如，控制器可能用缓存中的缓存单元大小作为块大小，这可能有浪费。控制器也可能把块大小、缓存大小、读取单元的大小（在一个操作中读取的数据量）匹配起来。如果读的单位太大，RAID 缓存可能不太有效，最终可能会读取比真正需要的更多的数据，即使是微小的请求。

当然，在实践中很难知道是否有数据会跨越多个驱动器。即使块大小为 16 KB，与 InnoDB 的页大小相匹配，也不能让所有的读取对齐 16 KB 的边界。文件系统可能会把文件分成片段，每个片段的大小通常与文件系统的块大小对齐，大部分文件系统的块大小为 4KB。一些文件系统可能更聪明，但不应该指望它。

可以配置系统以便从应用到底层存储所有的块都对齐：InnoDB 的块、文件系统的块、LVM，分区偏移、RAID 条带、磁盘扇区。我们的基准测试表明，当一切都对齐时，随机读和随机写的性能可能分别提高 15% 和 23%。对齐一切的精密技术太特殊了，无法在这细说，但其他很多地方有很多不错的信息，包括我们的博客，*http://www.mysqlperformanceblog.com*。

RAID 缓存

RAID 缓存就是物理安装在 RAID 控制器上的（相对来说）少量内存。它可以用来缓冲硬盘和主机系统之间的数据。下面是 RAID 卡使用缓存的几个原因：

缓存读取

控制器从磁盘读取数据并发送到主机系统后，通过缓存可以存储读取的数据，如果将来的请求需要相同的数据，就可以直接使用而无须再次去读盘。

这实际上是 RAID 缓存一个很糟糕的用法。为什么呢？由于操作系统和数据库服务器有自己更大得多的缓存。如果数据在这些上层缓存中命中了，RAID 缓存中的数据就不会被使用。相反，如果上层的缓存没有命中，就有可能在 RAID 缓存中命中，
421 但这个概率是微乎其微的。因为 RAID 缓存要小得多，几乎肯定会被刷新掉，被其他数据填上去了。无论哪种方式，缓冲读都是浪费 RAID 缓存的事。

缓存预读数据

如果 RAID 控制器发现连续请求的数据，可能会决定做预读操作——就是预先取出估计很快会用到的数据。在数据被请求之前，必须有地方放这些数据。这也会使用 RAID 缓存来放。预读对性能的影响可能有很大的不同，应该检查确保预读确实有帮助。如果数据库服务器做了自己的智能预读（例如 InnoDB 的预读），RAID 控制器的预读可能就没有帮助，甚至可能会干扰所有重要的缓冲和同步写入。

缓冲写入

RAID 控制器可以在高速缓存里缓冲写操作，并且一段时间后再写到硬盘。这样做有双重的好处：首先，可以快得多地返回给主机系统写“成功”的信号，远远比写入到物理磁盘上要快；其次，可以通过积累写操作从而更有效地批量操作[注22]。

内部操作

某些 RAID 的操作是非常复杂的——尤其是 RAID 5 的写入操作，其中要计算校验位，用来在发生故障时重建数据。控制器做这类内部操作需要使用一些内存。

这也是 RAID 5 在一些 RAID 控制器上性能差的原因：为了好的性能需要读取大量数据到内存。有些控制器不能较好地平衡缓存写和 RAID 5 校验位操作所需要的内存。

一般情况下，RAID 控制器的内存是一种稀缺资源，应该尽量用在刀刃上。缓存读取通常是一种浪费，但是缓冲写入是加速 I/O 性能的一个重要途径。许多控制器可以选择如何分配内存。例如，可以选择有多少缓存用于写入和多少用于读取。对于 RAID 0、RAID 1 和 RAID 10，应该把控制器缓存 100% 分配给写入用。对于 RAID 5，应该保留一些内存给内部操作。通常这是正确的建议，但并不总是适用——不同的 RAID 卡需要不同的配置。

当正在用 RAID 缓存缓冲写入时，许多控制器可以配置延迟写入多久时间（例如一秒钟、五秒钟，等等）是可以接受的。较长的延迟意味着更多的写入可以组合在一起更有效地刷新到磁盘。缺点是写入会变得更加“突发的”。但这不是一件坏事，除非应用一连串的写请求把控制器的缓存填满了才被刷新到磁盘。如果没有足够的空间存放应用程序的写入请求，写操作就会等待。保持短的延迟意味着可以有更多的写操作，并且会更低效[注23]，但能抚平性能波动，并且有助于保持更多的空闲缓存，来接收应用程序的爆发请求。（我们在这里简化了——事实上控制器往往很复杂，不同的供应商有自己的均衡算法，所以我们只是试图覆盖基本原则。） 422

写入缓冲对同步写入非常有用，例如事务日志和二进制日志（sync_binlog 设置为 1）调用的 fsync()，但是除非控制器有电池备份单元（BBU）或其他非易失性存储[注24]，否

注 22： 可以缓冲随机 I/O 部分合并为顺序 I/O。——译者注

注 23： 因为没有充分地合并 I/O。——译者注

注 24： 有几种技术，包括电容器和闪存存储，但这里我们都归结到 BBU 这一类。。

则不应该启用 RAID 缓存。不带 BBU 的情况下缓冲写，在断电时，有可能损坏数据库，甚至是事务性文件系统。然而，如果有 BBU，启用写入缓存可以提升很多日志刷新的工作的性能，例如事务提交时刷新事务日志。

最后要考虑的是，许多硬盘驱动器有自己的缓存，可能有“假”的 fsync() 操作，欺骗 RAID 控制器说数据已被写入物理介质[注25]。有时可以让硬盘直接挂载（而不是挂到 RAID 控制器上），让操作系统管理它们的缓存，但这并不总是有效。这些缓存通常在做 fsync() 操作时被刷新，另外同步 I/O 也会绕过它们直接访问磁盘，但是再次提醒，硬盘驱动器可能会骗你。应该确保这些缓存在 fsync() 时真的刷新了，否则就禁用它们，因为磁盘缓存没有电池供电（所以断电会丢失）。操作系统或 RAID 固件没有正确地管理硬盘管理已经造成了许多数据丢失的案例。

由于这个以及其他原因，当安装新硬件时，做一次真实的宕机测试（比如拔掉电源）是很有必要的。通常这是找出隐藏的错误配置或者诡异的硬盘问题的唯一办法。在 *http://brad.livejournal.com/2116715.html* 有个方便的脚本可以使用。

为了测试是否真的可以依赖 RAID 控制器的 BBU，必须像真实情况一样切断电源一段时间，因为某些单元断电超过一段时间后就可能会丢失数据。这里再次重申，任何一个环节出现问题都会使整个存储组件失效。

9.7 SAN 和 NAS

SAN（Storage Area Network）和 NAS（Network-Attached Storage）是两个外部文件存储设备加载到服务器的方法。不同的是访问存储的方式。访问 SAN 设备时通过块接口，服务器直接看到一块硬盘并且可以像硬盘一样使用，但是 NAS 设备通过基于文件的协议来访问，例如 NFS 或 SMB。SAN 设备通常通过光纤通道协议（FCP）或 iSCSI 连接到服务器，而 NAS 设备使用标准的网络连接。还有一些设备可以同时通过这两种方式访问，比如 NetApp Filer 存储系统。

423 在接下来的讨论中，我们将把这两种类型的存储统一称为 SAN。在后面的阅读应该记住这一点。主要区别在于作为文件还是块设备访问存储。

SAN 允许服务器访问非常大量的硬盘驱动器——通常在 50 块以上——并且通常配置大容量智能高速缓存来缓冲写入。块接口在服务器上以逻辑单元号（LUN）或者虚拟卷（除非使用 NFS）出现。许多 SAN 也允许多节点组成集群来获得更好的性能或者增加存储容量。

注 25： 就是 fsync 只是刷新到了硬盘上的缓存，这个缓存是没有电池的，所以掉电会丢失数据。——译者注

目前新一代 SAN 跟几年前的不同。许多新的 SAN 混合了闪存和机械硬盘，而不仅仅是机械硬盘了。它们往往有大到 TB 级或以上的闪存作为缓存，不像旧的 SAN，只有相对较小的缓存。此外，旧的 SAN 无法通过配置更大的缓存层来“扩展缓冲池”，而新的 SAN 有时可以。因此，相比之下新的 SAN 可以提供更好的性能。

9.7.1 SAN 基准测试

我们已经测试了多个 SAN 厂商的多种产品。表 9-3 展示了一些低并发场景下的典型测试结果。

表9-3：以16KB为单位同步单线程操作单个4GB文件的IOPS

设备	顺序写	顺序读	随机写	随机读
SAN1 做了 RAID 5	2428	5794	629	258
SAN1 做了 RAID 10	1765	3427	1725	213
SAN2 通过 NFS	1768	3154	2056	166
10k RPM 硬盘，RAID 1	7027	4773	2302	310
Intel SSD	3045	6266	2427	4397

具体的 SAN 厂商名字和配置做了保密处理，但是可以透露的是这些都不是便宜的 SAN。测试都是用同步的 16 KB 操作，模拟 InnoDB 配置在 O_DIRECT 模式时的操作方式。

从表 9-3 中可以得出什么结论？我们测试的系统不是都可以直接比较的，所以盯着这些好看的数据点来看不能客观地做出评价。然而，这些结果很好地说明了这类设备的总体性能表现，SAN 可以承受大量的连续写入，因为可以缓冲并合并 I/O。SAN 提供顺序读取没有问题，因为可以做预读并从缓存中提出数据。在随机写上会慢一些，因为写入操作不能较好地合并。因为读取通常在缓存中无法命中，必须等待硬盘驱动器响应，所以 SAN 很不适合做随机读取。最重要的是，服务器和 SAN 之间有传输延迟。这是为什么 424 通过 NFS 连接 SAN 时，提供的每秒随机读还不如一块本地磁盘的原因。

我们已经用较大尺寸的文件做了基准测试，但没有用其他尺寸的文件在上述的系统中测试。然而，无论结果如何，可以预见的是：不管多么强大的 SAN，对于小的随机操作，都无法获得良好的响应时间和吞吐量。延时的大部分都是由于服务器和 SAN 之间的链路导致的。

我们的基准测试显示的每秒操作吞吐量，并没有说出完整的故事。至少有三个重要指标：每秒吞吐量字节数、并发性和响应时间。在一般情况下，相对于直接连接存储（DAS），SAN 无论读写都可以提供更好的顺序 I/O 吞吐量。大多数 SAN 可以支持大量的并发性，但基准测试只有一个线程，这可以说明最坏的情况。但是，当工作集不能放到 SAN 的缓存时，随机读在吞吐量和延迟方面将变得很差，甚至延迟将高于直接访问本地存储。

9.7.2 使用基于 NFS 或 SMB 的 SAN

某些 SAN，例如 NetApp Filer 存储，通常通过 NFS 访问，而不是通过光纤或者 iSCSI。这曾经是我们希望避免的情况，但是 NFS 今天比以前好了很多。通过 NFS 可以获得相当好的性能，尽管需要专门配置网络。SAN 厂商提供的最佳实践指导可以帮助了解怎样配置。

主要考虑的事情是 NFS 协议自身怎样影响性能。许多文件元信息操作，通常在本地文件系统或者 SAN 设备(非 NAS)的内存中执行,但是在 NAS 上可能需要一次网络来回发送。例如，我们提醒过把二进制日志存在 NFS 上会损害服务器性能，即使关闭 sync_binlog 也无济于事。

也可以通过 SMB 协议访问 SAN 或者 NAS，需要考虑的问题类似：可能有更多的网络通信，会对延迟产生影响。对传统桌面用户这没什么影响，他们通常只是在挂载的驱动器上存储电子表格或者其他文档，或者只是为了备份复制一些东西到另一台服务器。但是用作 MySQL 读写它的文件，就会有严重的性能问题。

9.7.3 MySQL 在 SAN 上的性能

I/O 基准测试只是一种观察的方式,MySQL 在 SAN 上具体性能表现如何？在许多情况下，MySQL 运行得还可以，可以避免很多 SAN 可能导致性能下降的情况。仔细地做好逻辑
425 和物理设计，包括索引和适当的服务器硬件（尽量配置更多的内存！）可避免很多的随机 I/O 操作，或者可以转化为顺序的 I/O。然而，应该知道的是，通过一段时间的运行，这种系统可以达到一个微妙的平衡——引入一个新的查询,Schema 的变化或频繁的操作，都很容易扰乱这种平衡。

例如，一个 SAN 用户，我们知道他对每天的性能表现非常满意，直到有一天他想清理一张变得非常大的旧表中的大量数据行。这会导致一个长时间运行的 DELETE 语句，每秒只能删几百行，因为删除每行所需的随机 I/O，SAN 无法有效快速地执行。有没有办法来加快操作，它只是要花费很长的时间才能完成。另一个让他大吃一惊的事是，当对一个大表执行 ALTER 类似的操作时明显速度减慢。

这些都是些典型的例子,哪些工作放在 SAN 上不合适:执行大量的随机 I/O 的单线程任务。在当前版本的 MySQL 中，复制是另一个单线程任务。因此，备库的数据存储在 SAN 上，可能更容易落后于主库。批处理作业也可能运行得更慢。在非高峰时段或周末执行一个一次性的延迟敏感的操作是可以的，但是服务器的很多部分依然需要很好的性能，例如拷贝、二进制日志，以及 InnoDB 的事务日志上总是需要很好的小随机 I/O 性能。

9.7.4 应该用 SAN 吗

嗯，这是个长期存在的问题——在某些情况下，数百万美元的问题。有很多因素要考虑，以下我们列出其中的几个。

备份

集中存储使备份更易于管理。当所有数据都存储在一个地方时，可以只备份 SAN，只要确保已经确认过了所有的数据都在。这简化了问题，例如"你确定我们要备份所有的数据吗？"此外，某些设备有如连续数据保护（CDP）以及强大的快照功能等功能，使得备份更容易、更灵活。

简化容量规划

不确定需要多大容量吗？SAN 可以提供这种能力——购买大容量存储、分享给很多应用，并且可以调整大小并按需求重新发布。

存储整合还是服务器整合

某些 CIO 盘点数据中心运行了哪些东西时，可能会得出结论说大量的 I/O 容量被浪费了，这是把存储空间和 I/O 容量混为一谈了。毫无疑问的是，如果集中存储可以确保更好地利用存储资源，但这样做将会如何影响使用存储的系统？典型的数据库操作在性能上可以达到数量级的差异，因此可能会发现，如果集中存储可能需要增加 10 倍的服务器（或更多）才能处理原来的工作。尽管数据中心的 I/O 容量在 SAN 426
上可以更好地被利用，但是会导致其他系统无法充分被利用（例如数据库服务器花费大量时间等待 I/O、应用程序服务器花费大量时间等待数据库，依此类推）。在现实中我们已经看到过很多通过分散存储来整合服务器并削减成本的例子。

高可用

有时人们认为 SAN 是高可用解决方案。之所以会这样认为，可能是因为对高可用的真实含义的理解出现了分歧，我们将在第 12 章给出建议。

根据我们的经验，SAN 经常与故障和停机联系在一起，这不是因为它们不可靠——它们没什么问题，也确实很少出故障——只是因为人们都不愿意相信这样的工程奇迹其实也会坏的，因而缺乏这方面的准备。此外，SAN 有时是一个复杂的、神秘的黑盒子，当出问题的时候没有人知道该如何解决，并且价格昂贵，难以快速构建管理 SAN 所需的专业知识。大多数的 SAN 都对外缺乏可见性（就是个黑盒子），这也是为什么不应该只是简单地信任 SAN 管理员、支持人员或管理控制台的原因。我们看到过所有这三种人都错了的情况：当 SAN 出了问题，如出现硬盘驱动器故障导致性能下降[注26]的案例。这是另一个推荐使用 *sysbench* 的理由：*sysbench* 可以快速地完成一个 I/O 基准测试以证明是否是 SAN 的问题。

注 26： 基于网络的 SAN 管理控制台坚持所有硬盘驱动器是健康的——直到我们要求管理员按 Shift+F5 来禁用他的浏览器缓存并强制刷新控制台！

服务器之间的交互

共享存储可能会导致看似独立的系统实际上是相互影响的，有时甚至会很严重。例如，我们知道一个 SAN 用户有个很粗放的认识，当开发服务器上有 I/O 密集型操作时，会引起数据库服务器几乎陷于停顿。批处理作业、ALTER TABLE、备份——任何一个系统上产生大量的 I/O 操作都可能会导致其他系统的 I/O 资源不足。有时的影响远远比直觉想象的糟糕，一个看似不起眼的操作可能会导致严重的性能下降。

成本

成本是什么？管理和行政费用？每秒 I/O 操作数（IOPS）中每个 I/O 操作的成本？标价？

有充分的理由使用 SAN，但无论销售人员说什么，至少从 MySQL 需要的性能类型来看，SAN 不是最佳的选择。（选择一个 SAN 供应商并跟它们的销售谈，你可能听到他们一般也是同意的，然后告诉你他们的产品是一个例外。）如果考虑性价比，结

427 论会更加清楚，因为闪存存储或配置有电池支持写缓存的 RAID 控制器加上老式硬盘驱动器，可以在低得多的价格下提供更好的性能。

关于这个话题，不要忘了让销售给你两台 SAN 的价格。至少需要两台，否则这台昂贵的 SAN 可能会成为故障中的单点。

有许多"血泪史"可以引以为戒，这不是试图吓唬你远离 SAN。我们知道的 SAN 用户都非常地爱这些存储！如果正在考虑是否使用 SAN，最重要的事情是想清楚要解决什么问题。SAN 可以做很多事情，但解决性能问题只是其中很小的一部分。相比之下，当不要求很多高性能的随机 I/O，但是对某些功能感兴趣的话，如快照、存储整合、重复数据删除和虚拟化，SAN 可能非常适合。

因此，大多数 Web 应用不应该让数据库使用 SAN，但 SAN 在所谓的企业级应用很受欢迎。企业通常不太受预算限制，所以能够负担得起作为"奢侈品"的 SAN。（有时 SAN 甚至作为一种身份的象征！）

9.8 使用多磁盘卷

我们迟早都会碰到文件应该放哪的问题，因为 MySQL 创建了多种类型的文件：

- 数据和索引文件
- 事务日志文件
- 二进制日志文件
- 常规日志（例如，错误日志、查询日志和慢查询日志）
- 临时文件和临时表

MySQL 没有提供复杂的空间管理功能。默认情况下，只是简单地把每个 Schema 的文件放入一个单独的目录。有少量选项来控制数据文件放哪。例如，可以指定 MyISAM 表的索引位置，也可以使用 MySQL 5.1 的分区表。

如果正在使用 InnoDB 默认配置，所有的数据和文件都放在一组数据文件（共享表空间）中，只有表定义文件放在数据目录。因此，大部分用户把所有的数据和文件放在了单独的卷。

然而，有时使用多个卷可以帮助解决 I/O 负载高的问题。例如，一个批处理作业需要写入很多数据到一张巨大的表，将这张表放在单独的卷上，可以避免其他查询的 I/O 受到影响。理想的情况下，应该分析不同数据的 I/O 访问类型，才能把数据放在适当的位置，但这很难做到，除非已经把数据放在不同的卷上。

你可能已经听过标准建议，就是把事务日志和数据文件放在不同的卷上面，这样日志的顺序 I/O 和数据的随机 I/O 不会互相影响。但是除非有很多硬盘（20 或更多）或者闪存存储，否则在这样做之前应该考虑清楚代价。 428

二进制日志和数据文件分离的真正的优势，是减少事故中同时丢失数据和日志文件的可能性。如果 RAID 控制器上没有电池支持的写缓存，把它们分开是很好的做法。

但是，如果有备用电池单元，分离卷就可能不是想象中那么必要了。性能差异很小是主要原因。这是因为即使有大量的事务日志写入，其中大部分写入都很小。因此，RAID 缓存通常会合并 I/O 请求，通常只会得到每秒的物理顺序写请求。这通常不会干预数据文件的随机 I/O，除非 RAID 控制器真的整体上饱和了。一般的日志，其中有连续的异步写入、负荷也低，可以较好地与数据分享一个卷。

将日志放在独立的卷是否可以提升性能？通常情况下是的，但是从成本的角度来看这个问题，是否真的值得这么做，答案往往是否定的，尽管很多人不这么认为。

原因是：为事务日志提供专门的硬盘是很昂贵的。假设有六个硬盘驱动器，最常规的做法是把所有六块盘放到一个 RAID 卷，或者分成两部分，四个放数据，两个放事务日志。不过如果这样做，就减少了三分之一的硬盘放数据文件，这会导致性能显著地下降。此外，专门提供两个驱动器，对负载的影响也微不足道（假设 RAID 控制器有电池支持的写缓存）。

另一方面，如果有很多硬盘，投入一些给事务日志可能会从中受益。例如，一共有 30 块硬盘，可以分两块硬盘（配置为一个 RAID 1 的卷）给日志，能让日志写尽可能快。对于额外的性能，也可以在 RAID 控制器中分配一些写缓存空间给这个 RAID 卷。

成本效益不是唯一考虑的因素。可能想保持 InnoDB 的数据和事务日志在同一个卷的另一个原因是，这种策略可以使用 LVM 快照做无锁的备份。某些文件系统允许一致的多卷快照，并且对这些文件系统，这是一个很轻量的操作，但对于 ext3 有很多东西需要注意。(也可以使用 Percona XtraBackup 来做无锁备份，关于此主题更多的信息，请参阅第 15 章)

如果已经启用 sync_binlog，二进制日志在性能方面与事务日志相似了。然而，二进制日志存储跟数据放在不同的卷，实际上是一个好主意——把它们分开存放更安全，因此即使数据丢失，二进制日志也可以保存下来。这样，可以使用二进制日志做基于时间点的恢复。这方面的考虑并不适用于 InnoDB 的事务日志，因为没有数据文件，它们就没
429 用了，你不能将事务日志应用到昨晚的备份。(事务日志和二进制日志之间的区别在其他数据库的 DBA 看来，很难搞明白，在其他数据库这就是同一个东西。)

另外一个常见的场景是分离出临时目录的文件，MySQL 做 filesorts（文件排序）和使用磁盘临时表时会写到临时目录。如果这些文件不会太大的话，最好把它们放在临时内存文件系统，如 *tmpfs*。这是速度最快的选择。如果在你的系统上这不可行，就把它们放在操作系统盘上。

典型的磁盘布局是有操作系统盘、交换分区和二进制日志的盘，它们放在 RAID 1 卷上。还要有一个单独的 RAID 5 或 RAID 10 卷，放其他的一切东西。

9.9 网络配置

就像延迟和吞吐量是硬盘驱动器的限制因素一样，延迟和带宽（实际上和吞吐量是同一回事）也是网络连接的限制因素。对于大多数应用程序来说，最大的问题是延时。典型的应用程序都需要传输很多很小的网络包，并且每次传输的轻微延迟最终会被累加起来。

运行不正常的网络通常也是主要的性能瓶颈之一。丢包是一个普遍存在的问题。即使 1% 的丢包率也足以造成显著的性能下降，因为在协议栈的各个层次都会利用各种策略尝试修复问题，例如等待一段时间再重新发送数据包，这就增加了额外的时间。另一个常见的问题是域名解析系统（DNS）损坏或者变慢了。

DNS 足以称为“阿基里斯之踵”，因此在生产服务器上启用 skip_name_resolve 是个好主意。损坏或缓慢的 DNS 解析对许多应用程序都是个问题，对 MySQL 尤为严重。当 MySQL 收到一个连接请求时，它同时需要做正向和反向 DNS 查找。有很多原因可能导致这个过程出问题。当问题出现时，会导致连接被拒绝，或者使得连接到服务器的过程变慢，这通常都会造成严重的影响，甚至相当于遭遇了拒绝服务攻击（DDOS）。如果启用 skip_name_resolve 选项，MySQL 将不会做任何 DNS 查找的工作。然而，这也意味着，用户账户必须在 host 列使用具有唯一性的 IP 地址，“localhost”或者 IP 地址通

配符。那些在 host 列使用主机名的用户账户都将不能登录。

典型的 Web 应用中另一个常见的问题来源是 TCP 积压，可以通过 MySQL 的 back_log 选项来配置。这个选项控制 MySQL 的传入 TCP 连接队列的大小。在每秒有很多连接创建和销毁的环境中，默认值 50 是不够的。设置不够的症状是，客户端会看到零星的“连接被拒绝”的错误，配以三秒超时规则。在繁忙的系统中这个选项通常应加大。把这个选项增加到数百甚至数千，似乎没有任何副作用，事实上如果你看得远一些，可能还需 430
要配置操作系统的 TCP 网络设置。在 GNU / Linux 系统，需要增加 somaxconn 限制，默认只有 128，并且需要检查 *sysctl* 的 tcp_max_syn_back_log 设置（在本节稍后有一个例子）。

应该设计性能良好的网络，而不是仅仅接受默认配置的性能。首先，分析节点之间有多少跳跃点，以及物理网络布局之间的映射关系。例如，假设有 10 个网页服务器，通过千兆以太网（1 GigE）连接到“网页”交换机，这个交换机也通过千兆网络连接到“数据库”交换机。如果不花时间去追踪连接，可能不会意识到从所有数据库服务器到所有网页服务器的总带宽是有限的！并且每次跨越交换机都会增加延时。

监控网络性能和所有网络端口的错误是正确的做法，要监控服务器、路由器和交换机的每个端口。多路由流量绘图器（Multi Router Traffic Grapher），或者说 MRTG（*http://oss.oetiker.ch/mrtg/*），对设备监控而言是个靠得住的开源解决方案。其他常见的网络性能监控工具（与设备监控不同）还有 Smokeping（*http://oss.oetiker.ch/smokeping/*）和 Cacti（*http://www.cacti.net*）。

网络物理隔离也是很重要的因素。城际网络相比数据中心的局域网的延迟要大得多，即使从技术上来说带宽是一样的。如果节点真的相距甚远，光速也会造成影响。例如，在美国的西部和东部海岸都有数据中心，相隔约 3 000 公里。光的速度是 186 000 米每秒，因此一次通信不可能低于 16 毫秒，往返至少需要 32 毫秒。物理距离不仅是性能上的考虑，也包括设备之间通信的考虑。中继器、路由器和交换机，所有的性能都会有所降级。再次，越广泛地分隔开的网络节点，连接的不可预知和不可靠因素越大。

尽可能避免实时的跨数据中心的操作是明智的[注27]。如果不可能做到这一点，也应该确保应用程序能正常处理网络故障。例如，我们不希望看到由于 Web 服务器通过丢包严重的网络连接远程的数据中心时，由于 Apache 进程挂起而新建了很多进程的情况发生。

在本地，请至少用千兆网络。骨干交换机之间可能需要使用万兆以太网。如果需要更大的带宽，可以使用网络链路聚合：连接多个网卡（NIC），以获得更多的带宽。链路聚合本质上是并行网络，作为高可用策略的一部分也很有帮助。

注 27： 复制不算实时跨数据中心操作，它不是实时的，并且通常把数据复制到一个远程位置有助于提升数据安全性（容灾）。我们下一章会更多覆盖这个内容。

431 如果需要非常高的吞吐量，也许可以通过改变操作系统的网络配置来提高性能。如果连接不多，但是有很多的查询和很大的结果集，则可以增加 TCP 缓冲区的大小。具体的实现依赖于操作系统，对于大多数的 GNU/ Linux 系统，可以改变 */etc/sysctl.conf* 中的值并执行 *sysctl -p*，或者使用 */proc* 文件系统写入一个新的值到 */proc/sys/net/* 里面的文件。搜索"TCP tuning guide"，可以找到很多好的在线教程。

然而，调整设置以有效地处理大量连接和小查询的情况通常更重要。比较常见的调整之一，就是要改变本地端口的范围。系统的默认值如下：

```
[root@server ~]# cat /proc/sys/net/ipv4/ip_local_port_range
32768   61000
```

有时你也许需要改变这些值，调整得更大一些。例如：

```
[root@server ~]# echo 1024 65535 > /proc/sys/net/ipv4/ip_local_port_range
```

如果要允许更多的连接进入队列，可以做如下操作：

```
[root@server ~]# echo 4096 > /proc/sys/net/ipv4/tcp_max_syn_backlog
```

对于只在本地使用的数据库服务器，对于连接端还未断开，但是通信已经中断的事件中使用的套接字，可以缩短 TCP 保持状态的超时时间。在大多数系统上默认是一分钟，这个时间太长了：

```
[root@server ~]# echo <value> > /proc/sys/net/ipv4/tcp_fin_timeout
```

这些设置大部分时间可以保留默认值。通常只有当发生了特殊情况，例如网络性能极差或非常大量的连接，才需要进行修改。在互联网上搜索"TCP variables"，可以发现很多不错的阅读资料，除了上面提到的，还能看到很多其他的变量。

9.10 选择操作系统

GNU/Linux 如今是高性能 MySQL 最常用的操作系统，但是 MySQL 本身可以运行在很多操作系统上。

Solaris 是 SPARC 硬件上的领跑者，在 x86 硬件上也可以运行。Solaris 常用在要求高可靠的应用上面。Solaris 在某些方面的易用性可能没有 GNU/Linux 的名气大，但确实是一个坚固的操作系统，包含许多先进的功能。尤其是 Solaris 10 增加了 ZFS 文件系统，包含了很多先进的故障排除工具（如 DTrace）、良好的多线程性能，以及称为 Solaris Zones 的虚拟化技术，有助于资源管理。

FreeBSD 是另一种选择。它历来与 MySQL 配合有一些问题，大多时候都涉及到线程支持，但新的版本要好得多。如今，看到 MySQL 在 FreeBSD 上大规模部署的场景并不是什么稀罕事。ZFS 也可以在 FreeBSD 上使用。 432

通常用于开发和桌面应用程序的 MySQL 选择的是 Windows。也有企业级的 MySQL 部署在 Windows 上，但一般的企业级 MySQL 更多的还是部署在类 UNIX 操作系统上。我们不希望引起任何有关操作系统的争论，需要指出的是在异构操作系统环境中使用 MySQL 是不存在问题的。在类 UNIX 的操作系统上运行的 MySQL 服务器，同时在 Windows 上运行 Web 服务器，然后通过高品质的 .NET 连接器（这是 MySQL 免费提供的）进行连接，这是一个非常合理的架构。从 UNIX 连接到 Windows 上的 MySQL 服务器和连接到另一台 UNIX 上的 MySQL 服务器一样简单。

当选择操作系统时，如果使用的是 64 位架构的硬件（见前面介绍的“CPU 架构”），请确保安装的是 64 位版本的操作系统。

谈到 GNU/Linux 发行版时，个人的喜好往往是决定性的因素。我们认为最好的策略是使用专为服务器应用程序设计的发行版，而不是桌面发行版。要考虑发行版的生命周期、发布和更新政策，并检查供应商的支持是否有效。红帽子企业版 Linux 是一个高品质、稳定的发行版；CentOS 是一个受欢迎的二进制兼容替代品（免费），但已经因为延后时间较长获得了一些批评；还有 Oracle 发行的 Oracle Enterprise Linux；另外，Ubuntu 和 Debian 也是流行的发行版。

9.11 选择文件系统

文件系统的选择非常依赖于操作系统。在许多系统中，如 Windows 就只有一到两个选择，而且只有一个（NTFS）真的是能用的。比较而言，GNU/Linux 则支持多种文件系统。

许多人想知道哪个文件系统在 GNU/Linux 上能提供最好的 MySQL 性能，或者更具体一些，哪个对 InnoDB 和 MyISAM 而言是最好的选择。实际的基准测试表明，大多数文件系统在很多方面都非常接近，但测试文件系统的性能确实是一件烦心事。文件系统的性能是与工作负载相关的，没有哪个文件系统是“银弹”。大部分情况下，给定的文件系统不会明显地表现得与其他文件系统不一样。除非遇到了文件系统的限制，例如，它怎么支持并发、怎么在多文件下工作、怎么对文件切片，等等。

要考虑的更重要的问题是崩溃恢复时间，以及是否会遇到特定的限制，如目录下有许多文件会导致运行缓慢（这是 ext2 和旧版本 ext3 下一个臭名昭著的问题，但当前版本的 ext3 和 ext4 中用 `dir_index` 选项解决了问题）。文件系统的选择对确保数据安全是非常重要的，所以我们强烈建议不要在生产系统做实验。

433 如果可能，最好使用日志文件系统，例如 ext3、ext4、XFS、ZFS 或者 JFS。如果不这么做，崩溃后文件系统的检查可能耗费相当长的时间。如果系统不是很重要，非日志文件系统性能可能比支持事务的好。例如，ext2 可能比 ext3 工作得好，或者可以使用 *tunefs* 关闭 ext3 的日志记录功能。挂载时间对某些文件系统也是一个因素。例如，ReiserFS，在一个大的分区上可能用很长时间来挂载和执行日志恢复。

如果使用 ext3 或者其继承者 ext4，有三个选项来控制数据怎么记日志，这可以放在 */etc/fstab* 中作为挂载选项：

`data=writeback`

: 这个选项意味着只有元数据写入日志。元数据写入和数据写入并不同步。这是最快的配置，对 InnoDB 来说通常是安全的，因为 InnoDB 有自己的事务日志。唯一的例外是，崩溃时恰好导致 *.frm* 文件损坏了。

  这里给出一个使用这个配置可能导致问题的例子。当程序决定扩展一个文件使其更大，元数据（文件大小）会在数据实际写到（更大的）文件之前记录并写下操作情况。结果就是文件的尾部——最新扩展的区域——会包含垃圾数据。

`data=ordered`

: 这个选项也只会记录元数据，但提供了一些一致性保证，在写元数据之前会先写数据，使它们保持一致。这个选项只是略微比 writeback 选项慢，但是崩溃时更安全。

  在此配置中，如果我们再次假设程序想要扩展一个文件，该文件的元数据将不能反映文件的新大小，直到驻留在新扩展区域中的数据被写到文件中了。

`data=journal`

: 此选项提供了原子日志的行为，在数据写入到最终位置之前将记录到日志中。这个选项通常是不必要的，它的开销远远高于其他两个选项。然而，在某些情况下反而可以提高性能，因为日志可以让文件系统延迟把数据写入最终位置的操作。

不管哪种文件系统，都有一些特定的选项最好禁用，因为它们没有提供任何好处，反而增加了很多开销。最有名的是记录访问时间的选项，甚至读文件或目录时也要进行一次写操作。在 */etc/fstab* 中添加 noatime、nodiratime 挂载选项可以禁用此选项，这样做有时可以提高 5% ~ 10% 的性能，具体取决于工作负载和文件系统（虽然在其他场景下差别可能不是太大）。下面是 */etc/fstab* 中的一个例子，对 ext3 选项做设置的行：

```
/dev/sda2 /usr/lib/mysql ext3 noatime,nodiratime,data=writeback 0 1
```

434 还可以调整文件系统的预读的行为，因为这可能也是多余的。例如，InnoDB 有自己的预读策略，所以文件系统的预读就是重复多余的。禁用或限制预读对 Solaris 的 UFS 尤其有利。使用 O_DIRECT 选项会自动禁用预读。

一些文件系统可能不支持某些需要的功能。例如，若让 InnoDB 使用 O_DIRECT 刷新方式，文件系统能支持 Direct I/O 是非常重要的。此外，一些文件系统处理大量底层驱动器比其他的文件系统更好，举例来说 XFS 在这方面通常比 ext3 好。最后，如果打算使用 LVM 快照来初始化备库或进行备份，应该确认选择的文件系统和 LVM 版本能很好地协同工作。

表 9-4 某些常见文件系统的特性总结。

表9-4：常见文件系统特性

文件系统	操作系统	支持日志	大目录
ext2	GNU/Linux	否	否
ext3	GNU/Linux	可选	可选 / 部分
ext4	GNU/Linux	是	是
HFS Plus	Mac OS	可选	是
JFS	GNU/Linux	是	否
NTFS	Windows	是	是
ReiserFS	GNU/Linux	是	是
UFS (Solaris)	Solaris	是	可调的
UFS (FreeBSD)	FreeBSD	否	可选 / 部分
UFS2	FreeBSD	否	可选 / 部分
XFS	GNU/Linux	是	是
ZFS	Solaris, FreeBSD	是	是

我们通常建议客户使用 XFS 文件系统。ext3 文件系统有太多严重的限制，例如 inode 只有一个互斥变量，并且 fsync() 时会刷新所有脏块，而不只是单个文件。很多人感觉 ext4 文件系统用在生产环境有点太新了，不过现在似乎正日益普及。

9.12 选择磁盘队列调度策略

在 GNU / Linux 上，队列调度决定了到块设备的请求实际上发送到底层设备的顺序。默认情况下使用 cfq（Completely Fair Queueing，完全公平排队）策略。随意使用的笔记本和台式机使用这个调度策略没有问题，并且有助于防止 I/O 饥饿，但是用于服务器则是有问题的。在 MySQL 的工作负载类型下，cfq 会导致很差的响应时间，因为会在队列中延迟一些不必要的请求。 435

可以用下面的命令来查看系统所有支持的以及当前在用的调度策略：

```
$ cat /sys/block/sda/queue/scheduler
noop deadline [cfq]
```

这里 *sda* 需要替换成想查看的磁盘的盘符。在我们的例子中，方括号表示正在使用的调度策略。cfq 之外的两个选项都适合服务器级的硬件，并且在大多数情况下，它们工作同样出色。noop 调度适合没有自己的调度算法的设备，如硬件 RAID 控制器和 SAN。deadline 则对 RAID 控制器和直接使用的磁盘都工作良好。我们的基准测试显示，这两者之间的差异非常小。重要的是别用 cfq，这可能会导致严重的性能问题。

不过这个建议也需要有所保留的，因为磁盘调度策略实际上在不同的内核有很多不一样的地方，千万不能望文生义。

9.13 线程

MySQL 每个连接使用一个线程，另外还有内部处理线程、特殊用途的线程，以及所有存储引擎创建的线程。在 MySQL 5.5 中，Oracle 提供了一个线程池插件，但目前尚不清楚在实际应用中能获得什么好处。

无论哪种方式，MySQL 都需要大量的线程才能有效地工作。MySQL 确实需要内核级线程的支持，而不只是用户级线程，这样才能更有效地使用多个 CPU。另外也需要有效的同步原子，例如互斥变量。操作系统的线程库必须提供所有的这些功能。

GNU/Linux 提供两个线程库：LinuxThreads 和新的原生 POSIX 线程库（NPTL）。LinuxThreads 在某些情况下仍然使用，但现在的发行版已经切换到 NPTL，并且大部分应用已经不再加载 LinuxThreads。NPTL 更轻量，更高效，也不会有那些 LinuxThreads 遇到的问题。

FreeBSD 会加载许多线程库。从历史上看，它对线程的支持很弱，但现在已经变得好多了，在一些测试中，甚至优于 SMP 系统上的 GNU/Linux。在 FreeBSD 6 和更新版本，推荐的线程库是 *libthr*，早期版本使用的 *linuxthreads*，是 FreeBSD 从 GNU/Linux 上移植的 LinuxThreads 库。

通常，线程问题都是过去的事了，现在 GNU/Linux 和 FreeBSD 都提供了很好的线程库。

436 Solaris 和 Windows 一直对线程有很好的支持，尽管直到 5.5 发布之前，MyISAM 都不能在 Windows 下很好地使用线程，但 5.5 里有显著的提升。

9.14 内存交换区

当操作系统因为没有足够的内存而将一些虚拟内存写到磁盘就会发生内存交换[注 28]。内存交换对操作系统中运行的进程是透明的。只有操作系统知道特定的虚拟内存地址是在物

注 28： 内存交换有时称为页面交换。从技术上来说，它们是不同的东西，但是人们通常把它们作为同义词。

理内存还是硬盘。

内存交换对 MySQL 性能影响是很糟糕的。它破坏了缓存在内存的目的，并且相对于使用很小的内存做缓存，使用交换区的性能更差。MySQL 和存储引擎有很多算法来区别对待内存中的数据和硬盘上的数据，因为一般都假设内存数据访问代价更低。

因为内存交换对用户进程不可见，MySQL（或存储引擎）并不知道数据实际上已经移动到磁盘，还会以为在内存中。

结果会导致很差的性能。例如，若存储引擎认为数据依然在内存，可能觉得为“短暂”的内存操作锁定一个全局互斥变量（例如 InnoDB 缓冲池 Mutex）是 OK 的。如果这个操作实际上引起了硬盘 I/O，直到 I/O 操作完成前任何操作都会被挂起。这意味着内存交换比直接做硬盘 I/O 操作还要糟糕。

在 GNU/Linux 上，可以用 *vmstat*（在下一部分展示了一些例子）来监控内存交换。最好查看 `si` 和 `so` 列报告的内存交换 I/O 活动，这比看 `swpd` 列报告的交换区利用率更重要。`swpd` 列可以展示那些被载入了但是没有被使用的进程，它们并不是真的会成为问题。我们喜欢 `si` 和 `so` 列的值为 0，并且一定要保证它们低于每秒 10 个块。

极端的场景下，太多的内存交换可能导致操作系统交换空间溢出。如果发生了这种情况，缺乏虚拟内存可能让 MySQL 崩溃。但是即使交换空间没有溢出，非常活跃的内存交换也会导致整个操作系统变得无法响应，到这种时候甚至不能登录系统去杀掉 MySQL 进程。有时当交换空间溢出时，甚至 Linux 内核都会完全 hang 住。

绝不要让系统的虚拟内存溢出！对交换空间利用率做好监控和报警。如果不知道需要多少交换空间，就在硬盘上尽可能多地分配空间，这不会对性能造成冲击，只是消耗了硬盘空间。有些大的组织清楚地知道内存消耗将有多大，并且内存交换被非常严格地控制，但是对于只有少量多用途的 MySQL 实例，并且工作负载也多种多样的环境，通常不切 437
实际。如果后者的描述更符合实际情况，确认给服务器一些“呼吸”的空间，分配足够的交换空间。

在特别大的内存压力下经常发生的另一件事是内存不足（OOM），这会导致踢掉和杀掉一些进程。在 MySQL 进程这很常见。在另外的进程上也挺常见，比如 SSH，甚至会让系统不能从网络访问。可以通过设置 SSH 进程的 `oom_adj` 或 `oom_score_adj` 值来避免这种情况。

可以通过正确地配置 MySQL 缓冲来解决大部分内存交换问题，但是有时操作系统的虚拟内存系统还是会决定交换 MySQL 的内存。这通常发生在操作系统看到 MySQL 发出了大量 I/O，因此尝试增加文件缓存来保存更多数据时。如果没有足够的内存，有些东

西就必须被交换出去，有些可能就是 MySQL 本身。有些老的 Linux 内核版本也有一些适得其反的优先级，导致本不应该被交换的被交换出去，但是在最近的内核都被缓解了。

有些人主张完全禁用交换文件。尽管这样做有时在某些内核拒绝工作的极端场景下是可行的，但这降低了操作系统的性能（在理论上不会，但是实际上会的）。同时这样做也是很危险的，因为禁用内存交换就相当于给虚拟内存设置了一个不可动摇的限制。如果 MySQL 需要临时使用很大一块内存，或者有很耗内存的进程运行在同一台机器（如夜间批量任务），MySQL 可能会内存溢出、崩溃，或者被操作系统杀死。

操作系统通常允许对虚拟内存和 I/O 进行一些控制。我们提供过一些 GNU/Linux 上控制它们的办法。最基本的方法是修改 */proc/sys/vm/swappiness* 为一个很小的值，例如 0 或 1。这告诉内核除非虚拟内存完全满了，否则不要使用交换区。下面是如何检查这个值的例子：

```
$ cat /proc/sys/vm/swappiness
60
```

这个值显示为 60，这是默认的设置（范围是 0 ~ 100）。对服务器而言这是个很糟糕的默认值。这个值只对笔记本适用。服务器应该设置为 0：

```
$ echo 0 > /proc/sys/vm/swappiness
```

另一个选项是修改存储引擎怎么读取和写入数据。例如，使用 `innodb_flush_method=O_DIRECT`，减轻 I/O 压力。Direct I/O 并不缓存，因此操作系统并不能把 MySQL 视为增加文件缓存的原因。这个参数只对 InnoDB 有效。你也可以使用大页，不参与换入换出。这对 MyISAM 和 InnoDB 都有效。

438 另一个选择是使用 MySQL 的 `memlock` 配置项，可以把 MySQL 锁定在内存。这可以避免交换，但是也可能带来危险：如果没有足够的可锁定内存，MySQL 在尝试分配更多内存时会崩溃。这也可能导致锁定的内存太多而没有足够的内存留给操作系统。

很多技巧都是对于特定内核版本的，因此要小心，尤其是当升级的时候。在某些工作负载下，很难让操作系统的行为合情合理，并且仅有的资源可能让缓冲大小达不到最满意的值。

9.15 操作系统状态

操作系统会提供一些帮助发现操作系统和硬件正在做什么的工具。在这一节，我们会展示一些例子，包括关于怎样使用两个最常用的工具——*iostat* 和 *vmstat*。如果系统不提供它们中的任何一个，有可能提供了相似的替代品。因此，我们的目的不是让大家熟练使

用 *iostat* 和 *vmstat*，而是告诉你用类似的工具诊断问题时应该看什么指标。

除了这些，操作系统也许还提供了其他的工具，如 *mpstat* 或者 *sar*。如果对系统的其他部分感兴趣，例如网络，你可能希望使用 *ifconfig*（除了其他信息，它能显示发生了多少次网络错误）或者 *netstat*。

默认情况下，*vmstat* 和 *iostat* 只是生成一个报告，展示自系统启动以来很多计数器的平均值，这其实没什么用。然而，两个工具都可以给出一个间隔参数，让它们生成增量值的报告，展示服务器正在做什么，这更有用。（第一行显示的是系统启动以来的统计，通常可以忽略这一行。）

9.15.1 如何阅读 vmstat 的输出

我们先看一个 *vmstat* 的例子。用下面的命令让它每 5 秒钟打印出一个报告：

```
$ vmstat 5
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 0  0   2632  25728  23176 740244    0    0   527   521   11     3 10  1 86  3
 0  0   2632  27808  23180 738248    0    0     2   430  222    66  2  0 97  0
```

可以用 Ctrl+C 停止 *vmstat*。可以看到输出依赖于所用的操作系统，因此可能需要阅读一下手册来解读报告。

刚启动不久，即使采用增量报告，第一行的值还是显示自系统启动以来的平均值，第二行开始展示现在正在发生的情况，接下来的行会展示每 5 秒的间隔内发生了什么。每一列的含义在头部，如下所示：

procs 439

r 这一列显示了多少进程正在等待 CPU，b 列显示多少进程正在不可中断地休眠（通常意味着它们在等待 I/O，例如磁盘、网络、用户输入，等等）。

memory

swpd 列显示多少块被换出到了磁盘（页面交换）。剩下的三个列显示了多少块是空闲的（未被使用）、多少块正在被用作缓冲，以及多少正在被用作操作系统的缓存。

swap

这些列显示页面交换活动：每秒有多少块正在被换入（从磁盘）和换出（到磁盘）。它们比监控 swpd 列重要多了。

大部分时间我们喜欢看到 si 和 so 列是 0，并且我们很明确不希望看到每秒超过 10 个块。突发性的高峰一样很糟糕。

io

这些列显示有多少块从块设备读取（bi）和写出（bo）。这通常反映了硬盘 I/O。

system

这些列显示了每秒中断（in）和上下文切换（cs）的数量。

cpu

这些列显示所有的 CPU 时间花费在各类操作的百分比，包括执行用户代码（非内核）、执行系统代码（内核）、空闲，以及等待 I/O。如果正在使用虚拟化，则第五个列可能是 st，显示了从虚拟机中“偷走”的百分比。这关系到那些虚拟机想运行但是系统管理程序转而运行其他的对象的时间。如果虚拟机不希望运行任何对象，但是系统管理程序运行了其他对象，这不算被偷走的 CPU 时间。

vmstat 的输出跟系统有关，所以如果看到跟我们展示的例子不同的输出，应该阅读系统的 vmstat(8) 手册。一个重要的提示是：内存、交换区，以及 I/O 统计是块数而不是字节。在 GNU/Linux，块大小通常是 1 024 字节。

440

9.15.2 如何阅读 iostat 的输出

现在让我们转移到 *iostat*[注29]。默认情况下，它显示了与 *vmstat* 相同的 CPU 使用信息。我们通常只是对 I/O 统计感兴趣，所以使用下面的命令只展示扩展的设备统计：

```
$ iostat -dx 5
Device:  rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda         1.6    2.8 2.5 1.8  138.8   36.9     40.7      0.1  23.2   6.0   2.6
```

与 *vmstat* 一样，第一行报告显示的是自系统启动以来的平均值（通常删掉它节省空间），然后接下来的报告显示了增量的平均值，每个设备一行。

有多种选项显示和隐藏列。官方文档有点难以理解，因此我们必须从源码中挖掘真正显示的内容是什么。说明的列信息如下：

rrqm/s 和 wrqm/s

每秒合并的读和写请求。“合并的”意味着操作系统从队列中拿出多个逻辑请求合并为一个请求到实际磁盘。

r/s 和 w/s

每秒发送到设备的读和写请求。

rsec/s 和 wsec/s

每秒读和写的扇区数。有些系统也输出为 rkB/s 和 wkB/s，意为每秒读写的千字节数。

注 29：我们本书展示的 *iostat* 的例子为了印刷被稍微重排了：我们减少了小数位来避免换行。我们是在 GNU/Linux 上展示例子。其他操作系统输出可能不完全一样。

为了简洁，我们省略了那些指标说明。

avgrq-sz

请求的扇区数。

avgqu-sz

在设备队列中等待的请求数。

await

磁盘排队上花费的毫秒数。很不幸，*iostat* 没有独立统计读和写的请求，它们实际上不应该被一起平均。当你诊断性能案例时这通常很重要。

svctm

服务请求花费的毫秒数，不包括排队时间。

%util 441

至少有一个活跃请求所占时间的百分比。如果熟悉队列理论中利用率的标准定义，那么这个命名很莫名其妙。它其实不是设备的利用率。超过一块硬盘的设备（例如 RAID 控制器）比一块硬盘的设备可以支持更高的并发，但是 `%util` 从来不会超过 100%，除非在计算时有四舍五入的错误。因此，这个指标无法真实反映设备的利用率，实际上跟文档说的相反，除非只有一块物理磁盘的特殊例子。

可以用 *iostat* 的输出推断某些关于机器 I/O 子系统的实际情况。一个重要的度量标准是请求服务的并发数。因为读写的单位是每秒而服务时间的单位是千分之一秒，所以可以利用利特尔法则（Little's Law）得到下面的公式，计算出设备服务的并发请求数[注30]：

```
concurrency = (r/s + w/s) * (svctm/1000)
```

这是一个 *iostat* 的输出示例：

```
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda        105     311 298 820   3236   9052       10      127   113     9    96
```

把数字带入并发公式，可以得到差不多 9.6 的并发性[注31]。这意味着在一个采样周期内，这个设备平均要服务 9.6 次的请求。例子来自于一个 10 块盘的 RAID 10 卷，所以操作系统对这个设备的并行请求运行得相当好。另一方面，这是一个出现串行请求的设备：

```
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sdc         81       0 280   0   3164      0       11         2     7     3    99
```

并发公式展示了这个设备每秒只处理一个请求。两个设备都接近于满负荷利用，但是它

注 30：另一种计算并发的方式是通过平均队列大小、服务时间，以及平均等待时间：(avuqu_sz * svctm) / await。

注 31：如果你做这个计算，会得到大约 10，因为为了格式化我们已经取整了 *iostat* 的输出。相信我们，确实是 9.6。

们的性能表现完全不一样。如果设备一直都像这些例子展示的一样忙，那么应该检查一下并发性，不管是不是接近于设备中的物理盘数，都需要注意。更低的值则说明有如文件系统串行的问题，就像我们前面讨论的。

9.15.3 其他有用的工具

我们展示 *vmstat* 和 *iostat* 是因为它们部署最广泛，并且 *vmstat* 通常默认安装在许多类
442 UNIX 操作系统上。然而，每种工具都有自身的限制，例如莫名奇妙的度量单位、当操作系统更新统计时取样间隔不一致，以及不能一次性看到所有重要的点。如果这些工具不能符合需求，你可能会对 *dstat*（*http://dag.wieers.com/home-made/dstat/*）或 *collectl*（*http://collectl.sourceforge.net*）感兴趣。

我们也喜欢用 *mpstat* 来观察 CPU 统计；它提供了更好的办法来观察 CPU 每个工作是如何进行的，而不是把它们搅在一块。有时在诊断问题时这非常重要。当分析硬盘 I/O 利用的时候，*blktrace* 可能也非常有用。

我们自己开发了 *iostat* 的替代品，叫做 *pt-diskstats*。这是 Percona Toolkit 的一部分。它解决了一些对 *iostat* 的抱怨，例如显示读写统计的方式，以及缺乏对并发量的可见性。它也是交互式的，并且是按键驱动的，所以可以放大缩小、改变聚集、过滤设备，以及显示和隐藏列。即使没有安装这个工具，也可以通过简单的 Shell 脚本来收集一些硬盘统计状态，这个工具也支持分析这样采集出来的文本。可以抓取一些硬盘活动样本，然后发邮件或者保存起来，稍后分析。实际上，我们第 3 章中介绍的 *pt-stalk*、*pt-collect*、和 *pt-sift* 三件套，都被设计得可以跟 *pt-diskstats* 很好地配合。

9.15.4 CPU 密集型的机器

CPU 密集型服务器的 *vmstat* 输出通常在 us 列会有一个很高的值，报告了花费在非内核代码上的 CPU 时钟；也可能在 sy 列有很高的值，表示系统 CPU 利用率，超过 20% 就足以令人不安了。在大部分情况下，也会有进程队列排队时间（在 r 列报告的）。下面是一个例子：

```
$ vmstat 5
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
 r  b   swpd   free   buff  cache    si    so    bi    bo   in    cs us sy id wa
10  2 740880  19256  46068 13719952   0     0  2788 11047 1423 14508 89  4  4  3
11  0 740880  19692  46144 13702944   0     0  2907 14073 1504 23045 90  5  2  3
 7  1 740880  20460  46264 13683852   0     0  3554 15567 1513 24182 88  5  3  3
10  2 740880  22292  46324 13670396   0     0  2640 16351 1520 17436 88  4  4  3
```

注意，这里也有合理数量的上下文切换（在 cs 列），除非每秒超过 100 000 次或更多，一般都不用担心上下文切换。当操作系统停止一个进程转而运行另一个进程时，就会产

生上下文切换。

例如，一查询语句在 MyISAM 上执行了一个非覆盖索引扫描，就会先从索引中读取元素，然后根据索引再从磁盘上读取页面。如果页面不在操作系统缓存中，就需要从磁盘进行物理读取，这就会导致上下文切换中断进程处理，直到 I/O 完成。这样一个查询可以导致大量的上下文切换。

如果我们在同一台机器观察 *iostat* 的输出（再次剔除显示启动以来平均值的第一行），可以发现磁盘利用率低于 50%： 443

```
$ iostat -dx 5
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda          0   3859  54 458   2063  34546       71        3     6     1    47
dm-0         0      0  54 4316  2063  34532        8       18     4     0    47
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda          0   2898  52 363   1767  26090       67        3     7     1    45
dm-0         0      0  52 3261  1767  26090        8       15     5     0    45
```

这台机器不是 I/O 密集型的，但是依然有相当数量的 I/O 发生，在数据库服务器中这种情况很少见。另一方面，传统的 Web 服务器会消耗大量 CPU 资源，但是很少发生 I/O，所以 Web 服务器的输出不会像这个例子。

9.15.5 I/O 密集型的机器

在 I/O 密集型工作负载下，CPU 花费大量时间在等待 I/O 请求完成。这意味着 *vmstat* 会显示很多处理器在非中断休眠（b 列）状态，并且在 wa 这一列的值很高，下面是个例子：

```
$ vmstat 5
procs -----------memory---------    ---swap-- -----io---- --system-- ----cpu----
 r  b   swpd   free   buff  cache    si   so    bi    bo   in    cs us sy id wa
 5  7 740632  22684  43212 13466436   0    0  6738 17222 1738 16648 19  3 15 63
 5  7 740632  22748  43396 13465436   0    0  6150 17025 1731 16713 18  4 21 58
 1  8 740632  22380  43416 13464192   0    0  4582 21820 1693 15211 16  4 24 56
 5  6 740632  22116  43512 13463484   0    0  5955 21158 1732 16187 17  4 23 56
```

这台机器的 *iostat* 输出显示硬盘一直很忙：[注 32]

```
$ iostat -dx 5
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda          0   5396 202  626  7319  48187       66       12    14     1   101
dm-0         0      0 202 6016  7319  48130        8       57     9     0   101
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda          0   5810 184  665  6441  51825       68       11    13     1   102
dm-0         0      0 183 6477  6441  51817        8       54     7     0   102
```

注 32： 在书的第二版中，我们混淆了“总是很忙”和“完全饱和”。总是在做事的硬盘并不总是达到极限，因为它们可能也能支持一些并发。

%util的值可能因为四舍五入的错误超过 100%。什么迹象意味着机器是 I/O 密集的呢？只要有足够的缓冲来服务写请求，即使机器正在做大量的写操作，也可能可以满足，但是却通常意味着硬盘可能会无法满足读请求。这听起来好象违反直觉，但是如果思考读和写的本质，就不会这么认为了：

444 • 写请求能够缓冲或同步操作。它们可以被我们本书讨论过的任意一层缓冲：操作系统层、RAID 控制器层，等等。
• 读请求就其本质而言都是同步的。当然程序可以猜测到可能需要某些数据，并异步地提前读取（预读）。无论如何，通常程序在继续工作前必须得到它们需要的数据。这就强制读请求为同步操作：程序必须被阻塞直到请求完成。

想想这种方式：你可以发出一个写请求到缓冲区的某个地方，然后过一会完成。甚至可以每秒发出很多这样的请求。如果缓冲区正确工作，并且有足够的空间，每个请求都可以很快完成，并且实际上写到物理硬盘是被重新排序后更有效地批量操作的。然而，没有办法对读操作这么做——不管多小或多少的请求，都不可能让硬盘响应说“这是你的数据，我等一会读它”。这就是为什么读需要 I/O 等待是可以理解的原因。

9.15.6 发生内存交换的机器

一台正在发生内存交换的机器可能在 swpd 列有一个很高的值，也可能不高。但是可以看到 si 和 so 列有很高的值，这是我们不希望看到的。下面是一台内存交换严重的机器的 *vmstat* 输出：

```
$ vmstat 5
procs -----------memory------------ ---swap---- -----io---- --system-- ----cpu----
 r  b    swpd   free   buff    cache    si    so    bi    bo   in    cs us sy id wa
 0 10 3794292  24436  27076 14412764 19853  9781 57874  9833 4084  8339  6 14 58 22
 4 11 3797936  21268  27068 14519324 15913 30870 40513 30924 3600  7191  6 11 36 47
 0 37 3847364  20764  27112 14547112   171 38815 22358 39146 2417  4640  6  8  9 77
```

9.15.7 空闲的机器

为完整起见，下面也给出一台空闲机器上的 *vmstat* 输出。注意，没有在运行或被阻塞的进程，idle 列显示 CPU 是 100% 的空闲。这个例子来源于一台运行红帽子企业版 Linux 5（RHEL 5）的机器，并且 st 列展示了从“虚拟机”偷来的时间。

```
$ vmstat 5
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy  id wa st
 0  0    108 492556   6768 360092    0    0   345   209    2   65  2  0  97  1  0
 0  0    108 492556   6772 360088    0    0     0    14  357   19  0  0 100  0  0
 0  0    108 492556   6776 360084    0    0     0     6  355   16  0  0 100  0  0
```

9.16 总结 445

为 MySQL 选择和配置硬件，以及根据硬件配置 MySQL，并不是什么神秘的艺术。通常，对于大部分目标需要的都是相同的技巧和知识。当然，也需要知道一些 MySQL 特有的特点。

我们通常建议大部分人在性能和成本之间找到一个好的平衡点。首先，出于多种原因，我们喜欢使用廉价服务器。举个例子，如果在使用服务器的过程中遇到了麻烦，并且在诊断时需要停止服务，或者希望只是简单地把出问题的服务器用另一台替换，如果使用的是一台 $5 000 的廉价服务器，肯定比使用一台超过 $50 000 或者更贵的服务器要简单得多。MySQL 通常也更适应廉价服务器，不管是从软件自身而言还是从典型的工作负载而言。

MySQL 需要的四种基本资源是：CPU、内存、硬盘以及网络资源。网络一般不会作为很严重的瓶颈出现，而 CPU、内存和磁盘通常是主要的瓶颈所在。对 MySQL 而言，通常希望有很多快速 CPU 可以用，但如果必须在快和多之间做选择，则一般会选择更快而不是更多（其他条件相同的情况下）。

CPU、内存以及磁盘之间的关系错综复杂，一个地方的问题可能会在其他地方显现出来。在对一个资源抛出问题时，问问自己是不是可能是由另外的问题导致的。如果遇到硬盘密集的操作，需要更多的 I/O 容量吗？或者是更多的内存？答案取决于工作集大小，也就是给定的时间内最常用的数据集。

在本书写作的过程中，我们觉得以下做法是合理的。首先，通常不要超过两个插槽。现在即使双路系统也可以提供很多 CPU 核心和硬件线程了，而且四路服务器的 CPU 要贵得多。另外，四路 CPU 的使用不够广泛（也就意味着缺少测试和可靠性），并且使用的是更低的时钟频率。最终，四路插槽的系统跨插槽的同步开销也显著增加。在内存方面，我们喜欢用价格经济的服务器内存。许多廉价服务器目前有 18 个 DIMM 槽，单条 8GB 的 DIMM 是最好的选择——每 GB 的价格与更低容量的 DIMM 相比差不多，但是比 16GB 的 DIMM 便宜多了。这是为什么我们今天看到很多服务器是 144GB 的内存的原因。这个等式会随着时间的变化而变化——可能有一天具有最佳性价比的是 16GB 的 DIMM，并且服务器出厂的内存槽数量也可能不一样——但是一般的原则还是一样的。

持久化存储的选择本质上归结为三个选项，以提高性能的次序排序：SAN、传统硬盘， 446
以及固态存储设备。

- 当需要功能和纯粹的容量时，SAN 是不错的。它们对许多工作负载都运行得不错，但缺点是很昂贵，并且对小的随机 I/O 操作有很大的延时，尤其是使用更慢的互联

方式（如 NFS）或工作集太大不足以匹配 SAN 内存的缓存时，延时会更大。要注意 SAN 的性能突变的情况，并且要非常小心避免灾难的场景。

- 传统硬盘很大，便宜，但是对随机读很慢。对大部分场景，最好的选择是服务器硬盘组成 RAID 10 卷。通常应该使用带有电池保护单元的 RAID 控制器，并且设置写缓存为 WriteBack 策略。这样一个配置对大部分工作负载都可以运行良好。
- 固态盘相对比较小并且昂贵，但是随机 I/O 非常快。一般分为两类：SSD 和 PCIe 设备。广泛地来说，SSD 更便宜，更慢，但缺少可靠性验证。需要对 SSD 做 RAID 以提升可靠性，但是大多数硬件 RAID 控制器不擅长这个任务[注33]。PCIe 设备很昂贵并且有容量限制，但是非常快并且可靠，而且不需要 RAID。

固态存储设备可以很大地提升服务器整体性能。有时候一个不算昂贵的 SSD，可以帮助解决经常在传统硬盘上遇到的特定工作负载的问题，如复制。如果真的需要很强的性能，应该使用 PCIe 设备。增加高速 I/O 设备会把服务器的性能瓶颈转移到 CPU，有时也会转移到网络。

MySQL 和 InnoDB 并不能完全发挥高端固态存储设备的性能，并且在某些场景下操作系统也不能发挥。但是提升依然很明显。Percona Server 对固态存储做了很多改进，并且很多改进在 5.6 发布时已经进入了 MySQL 主干代码。

对操作系统而言，只有很少的一些重要配置需要关注，大部分是关于存储、网络和虚拟内存管理的。如果像大部分 MySQL 用户一样使用 GNU/Linux，建议采用 XFS 文件系统，并且为服务器的页面交换倾向率（swapiness）和硬盘队列调度器设置恰当的值。有一些网络参数需要改变，可能还有一些其他的地方（例如禁用 SELinux）需要调优，但是前面说的那些改动的优先级应该更高一些。

注 33：有些 RAID 控制器对 SSD 支持很差，做了 RAID 性能下降。——译者注

第 10 章 447

复制

MySQL 内建的复制功能是构建基于 MySQL 的大规模、高性能应用的基础，这类应用使用所谓的“水平扩展”的架构。我们可以通过为服务器配置一个或多个备库[注1]的方式来进行数据同步。复制功能不仅有利于构建高性能的应用，同时也是高可用性、可扩展性、灾难恢复、备份以及数据仓库等工作的基础。事实上，可扩展性和高可用性通常是相关联的话题，我们会在接下来的三章详细阐述。

本章将阐述所有与复制相关的内容，首先简要介绍复制如何工作，然后讨论基本的复制服务搭建，包括与复制相关的配置以及如何管理和优化复制服务器。虽然本书的主题是高性能，但对于复制来说，我们同样需要关注其准确性和可靠性，因此我们也会讲述复制在什么情况下会失败，以及如何使其更好地工作。

10.1 复制概述

复制解决的基本问题是让一台服务器的数据与其他服务器保持同步。一台主库的数据可以同步到多台备库上，备库本身也可以被配置成另外一台服务器的主库。主库和备库之间可以有多种不同的组合方式。

MySQL 支持两种复制方式：基于行的复制和基于语句的复制。基于语句的复制（也称为逻辑复制）早在 MySQL 3.23 版本中就存在，而基于行的复制方式在 5.1 版本中才被加进来。这两种方式都是通过在主库上记录二进制日志[注2]、在备库重放日志的方式来实现异步的数据复制。这意味着，在同一时间点备库上的数据可能与主库存在不一致，并 448
且无法保证主备之间的延迟。一些大的语句可能导致备库产生几秒、几分钟甚至几个小时的延迟。

注 1： 可能有些地方将会复制备库（replica）称为从库（slave），这里我们尽量避免这种叫法。

注 2： 如果对二进制日志感到陌生，可以在第 8 章、本章剩下的部分以及第 15 章获得更多的信息。

MySQL 复制大部分是向后兼容的，新版本的服务器可以作为老版本服务器的备库，但反过来，将老版本作为新版本服务器的备库通常是不可行的，因为它可能无法解析新版本所采用的新的特性或语法，另外所使用的二进制文件的格式也可能不相同。例如，不能从 MySQL 5.1 复制到 MySQL 4.0。在进行大的版本升级前，例如从 4.1 升级到 5.0，或从 5.1 升级到 5.5，最好先对复制的设置进行测试。但对于小版本号升级，如从 5.1.51 升级到 5.1.58，则通常是兼容的。通过阅读每次版本更新的 ChangeLog 可以找到不同版本间做了什么修改。

复制通常不会增加主库的开销，主要是启用二进制日志带来的开销，但出于备份或及时从崩溃中恢复的目的，这点开销也是必要的。除此之外，每个备库也会对主库增加一些负载（例如网络 I/O 开销），尤其当备库请求从主库读取旧的二进制日志文件时，可能会造成更高的 I/O 开销。另外锁竞争也可能阻碍事务的提交。最后，如果是从一个高吞吐量（例如 5 000 或更高的 TPS）的主库上复制到多个备库，唤醒多个复制线程发送事件的开销将会累加。

通过复制可以将读操作指向备库来获得更好的读扩展，但对于写操作，除非设计得当，否则并不适合通过复制来扩展写操作。在一主库多备库的架构中，写操作会被执行多次，这时候整个系统的性能取决于写入最慢的那部分。

当使用一主库多备库的架构时，可能会造成一些浪费，因为本质上它会复制大量不必要的重复数据。例如，对于一台主库和 10 台备库，会有 11 份数据拷贝，并且这 11 台服务器的缓存中存储了大部分相同的数据。这和在服务器上有 11 路 RAID 1 类似。这不是一种经济的硬件使用方式，但这种复制架构却很常见，本章我们将讨论解决这个问题的方法。

10.1.1 复制解决的问题

下面是复制比较常见的用途：

数据分布

MySQL 复制通常不会对带宽造成很大的压力，但在 5.1 版本引入的基于行的复制会
449 比传统的基于语句的复制模式的带宽压力更大。你可以随意地停止或开始复制，并在不同的地理位置来分布数据备份，例如不同的数据中心。即使在不稳定的网络环境下，远程复制也可以工作。但如果为了保持很低的复制延迟，最好有一个稳定的、低延迟连接。

负载均衡

通过 MySQL 复制可以将读操作分布到多个服务器上，实现对读密集型应用的优化，并且实现很方便，通过简单的代码修改就能实现基本的负载均衡。对于小规模的应用，可以简单地对机器名做硬编码或使用 DNS 轮询（将一个机器名指向多个 IP 地

址）。当然也可以使用更复杂的方法，例如网络负载均衡这一类的标准负载均衡解决方案，能够很好地将负载分配到不同的 MySQL 服务器上。Linux 虚拟服务器（Linux Virtual Server，LVS）也能够很好地工作，第 11 章将详细地讨论负载均衡。

备份

对于备份来说，复制是一项很有意义的技术补充，但复制既不是备份也不能够取代备份。

高可用性和故障切换

复制能够帮助应用程序避免 MySQL 单点失败，一个包含复制的设计良好的故障切换系统能够显著地缩短宕机时间，我们将在第 12 章讨论故障切换。

MySQL 升级测试

这种做法比较普遍，使用一个更高版本的 MySQL 作为备库，保证在升级全部实例前，查询能够在备库按照预期执行。

10.1.2 复制如何工作

在详细介绍如何设置复制之前，让我们先看看 MySQL 实际上是如何复制数据的。总的来说，复制有三个步骤：

1. 在主库上把数据更改记录到二进制日志（Binary Log）中（这些记录被称为二进制日志事件）。
2. 备库将主库上的日志复制到自己的中继日志（Relay Log）中。
3. 备库读取中继日志中的事件，将其重放到备库数据之上。

以上只是概述，实际上每一步都很复杂，图 10-1 更详细地描述了复制的细节。

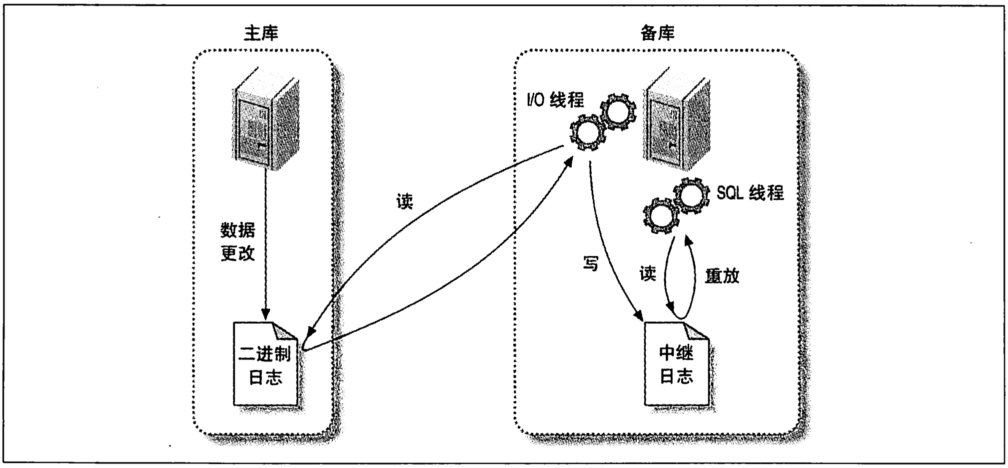


图10-1：MySQL复制如何工作

第一步是在主库上记录二进制日志（稍后介绍如何设置）。在每次准备提交事务完成数
450 据更新前，主库将数据更新的事件记录到二进制日志中。MySQL 会按事务提交的顺序而非每条语句的执行顺序来记录二进制日志。在记录二进制日志后，主库会告诉存储引擎可以提交事务了。

下一步，备库将主库的二进制日志复制到其本地的中继日志中。首先，备库会启动一个工作线程，称为 I/O 线程，I/O 线程跟主库建立一个普通的客户端连接，然后在主库上启动一个特殊的二进制转储（*binlog dump*）线程（该线程没有对应的 SQL 命令），这个二进制转储线程会读取主库上二进制日志中的事件。它不会对事件进行轮询。如果该线程追赶上了主库，它将进入睡眠状态，直到主库发送信号量通知其有新的事件产生时才会被唤醒，备库 I/O 线程会将接收到的事件记录到中继日志中。

> MySQL 4.0 之前的复制与之后的版本相比改变很大，例如 MySQL 最初的复制功能没有使用中继日志，所以复制只用到了两个线程，而不是现在的三个线程。目前大部分人都是使用的最新版本，因此在本章我们不会去讨论关于老版本复制的更多细节。

备库的 SQL 线程执行最后一步，该线程从中继日志中读取事件并在备库执行，从而实现备库数据的更新。当 SQL 线程追赶上 I/O 线程时，中继日志通常已经在系统缓存中，所以中继日志的开销很低。SQL 线程执行的事件也可以通过配置选项来决定是否写入其自己的二进制日志中，它对于我们稍后提到的场景非常有用。

图 10-1 显示了在备库有两个运行的线程，在主库上也有一个运行的线程：和其他普通连接一样，由备库发起的连接，在主库上同样拥有一个线程。

451 这种复制架构实现了获取事件和重放事件的解耦，允许这两个过程异步进行。也就是说 I/O 线程能够独立于 SQL 线程之外工作。但这种架构也限制了复制的过程，其中最重要的一点是在主库上并发运行的查询在备库只能串行化执行，因为只有一个 SQL 线程来重放中继日志中的事件。后面我们将会看到，这是很多工作负载的性能瓶颈所在。虽然有一些针对该问题的解决方案，但大多数用户仍然受制于单线程。

10.2 配置复制

为 MySQL 服务器配置复制非常简单。但由于场景不同，基本的步骤还是有所差异。最基本的场景是新安装的主库和备库，总的来说分为以下几步：

1. 在每台[注3]服务器上创建复制账号。
2. 配置主库和备库。
3. 通知备库连接到主库并从主库复制数据。

这里我们假定大部分配置采用默认值即可，在主库和备库都是全新安装并且拥有同样的数据（默认 MySQL 数据库）时这样的假设是合理的。接下来我们将展示如何一步步配置复制：假设有服务器 server1（IP 地址 192.168.0.1）和服务器 server2（IP 地址 192.168.0.2），我们将解释如何给一个已经运行的服务器配置备库，并探讨推荐的复制配置。

10.2.1 创建复制账号

MySQL 会赋予一些特殊的权限给复制线程。在备库运行的 I/O 线程会建立一个到主库的 TCP/IP 连接，这意味着必须在主库创建一个用户，并赋予其合适的权限。备库 I/O 线程以该用户名连接到主库并读取其二进制日志。通过如下语句创建用户账号：

```
mysql> GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.*
    -> TO repl@'192.168.0.%' IDENTIFIED BY 'p4ssword',;
```

我们在主库和备库都创建该账号。注意我们把这个账户限制在本地网络，因为这是一个特权账号（尽管该账号无法执行 select 或修改数据，但仍然能从二进制日志中获得一些数据）。

> 复制账户事实上只需要有主库上的 REPLICATION SLAVE 权限，并不一定需要每一端服务器都有 REPLICATION CLIENT 权限，那为什么我们要把这两种权限给主/备库都赋予呢？这有两个原因： 452
>
> - 用来监控和管理复制的账号需要 REPLICATION CLIENT 权限，并且针对这两种目的使用同一个账号更加容易（而不是为某个目的单独创建一个账号）。
> - 如果在主库上建立了账号，然后从主库将数据克隆到备库上时，备库也就设置好了——变成主库所需要的配置。这样后续有需要可以方便地交换主备库的角色。

10.2.2 配置主库和备库

下一步需要在主库上开启一些设置，假设主库是服务器 server1，需要打开二进制日志并指定一个独一无二的服务器 ID（server ID），在主库的 *my.cnf* 文件中增加或修改如下内容：

注 3： 严格来讲这不是必需的，但我们推荐这么做，稍后我们会解释为什么。

```
log_bin     = mysql-bin
server_id   = 10
```

实际取值由你决定，这里只是为了简单起见，当然也可以设置更多需要的配置。

必须明确地指定一个唯一的服务器 ID，默认服务器 ID 通常为 1（这和版本相关，一些 MySQL 版本根本不允许使用这个值）。使用默认值可能会导致和其他服务器的 ID 冲突，因此这里我们选择 10 来作为服务器 ID。一种通用的做法是使用服务器 IP 地址的末 8 位，但要保证它是不变且唯一的（例如，服务器都在一个子网里）。最好选择一些有意义的约定并遵循。

如果之前没有在 MySQL 的配置文件中指定 log-bin 选项，就需要重新启动 MySQL。为了确认二进制日志文件是否已经在主库上创建，使用 SHOW MASTER STATUS 命令，检查输出是否与如下的一致。MySQL 会为文件名增加一些数字，所以这里看到的文件名和你定义的会有点不一样。

```
mysql> SHOW MASTER STATUS;
+------------------+----------+--------------+------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+------------------+----------+--------------+------------------+
| mysql-bin.000001 |       98 |              |                  |
+------------------+----------+--------------+------------------+
1 row in set (0.00 sec)
```

备库上也需要在 *my.cnf* 中增加类似的配置，并且同样需要重启服务器。

453

```
log_bin           = mysql-bin
server_id         = 2
relay_log         = /var/lib/mysql/mysql-relay-bin
log_slave_updates = 1
read_only         = 1
```

从技术上来说，这些选项并不总是必要的。其中一些选项我们只是显式地列出了默认值。事实上只有 server_id 是必需的。这里我们同样也使用了 log_bin，并赋予了一个明确的名字。默认情况下，它是根据机器名来命名的，但如果机器名变化了可能会导致问题。为了简便起见，我们将主库和备库上的 log-bin 设置为相同的值。当然如果你愿意的话，也可以设置成别的值。

另外我们还增加了两个配置选项：relay_log（指定中继日志的位置和命名）和 log_slave_updates（允许备库将其重放的事件也记录到自身的二进制日志中），后一个选项会给备库增加额外的工作，但正如后面将会看到的，我们有理由为每个备库设置该选项。

有时候只开启了二进制日志，但却没有开启 log_slave_updates，可能会碰到一些奇怪的现象，例如，当配置错误时可能会导致备库数据被修改。如果可能的话，最好使用

read_only 配置选项，该选项会阻止任何没有特权权限的线程修改数据（所以最好不要给予用户超出需要的权限)。但 read_only 选项常常不是很实用，特别是对于那些需要在备库建表的应用。

> 不要在配置文件 *my.cnf* 中设置 master_port 或 master_host 这些选项，这是老的配置方式，已经被废弃，它只会导致问题，不会有任何好处。

10.2.3 启动复制

下一步是告诉备库如何连接到主库并重放其二进制日志。这一步不要通过修改 *my.cnf* 来配置，而是使用 CHANGE MASTER TO 语句，该语句完全替代了 *my.cnf* 中相应的设置，并且允许以后指向别的主库时无须重启备库。下面是开始复制的基本命令：

```
mysql> CHANGE MASTER TO MASTER_HOST='server1',
    -> MASTER_USER='repl',
    -> MASTER_PASSWORD='p4ssword',
    -> MASTER_LOG_FILE='mysql-bin.000001',
    -> MASTER_LOG_POS=0;
```

MASTER_LOG_POS 参数被设置为 0，因为要从日志的开头读起。当执行完这条语句后，可以通过 SHOW SLAVE STATUS 语句来检查复制是否正确执行。

454

```
mysql> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
             Slave_IO_State:
                Master_Host: server1
                Master_User: repl
                Master_Port: 3306
              Connect_Retry: 60
            Master_Log_File: mysql-bin.000001
        Read_Master_Log_Pos: 4
             Relay_Log_File: mysql-relay-bin.000001
              Relay_Log_Pos: 4
      Relay_Master_Log_File: mysql-bin.000001
           Slave_IO_Running: No
          Slave_SQL_Running: No
                             ...omitted...
      Seconds_Behind_Master: NULL
```

Slave_IO_State、Slave_IO_Running 和 Slave_SQL_Running 这三列显示当前备库复制尚未运行。聪明的读者可能已经注意到日志的开头是 4 而不是 0，这是因为 0 其实不是日志真正开始的位置，它仅仅意味着“在日志文件头”，MySQL 知道第一个事件从文件的第 4 位[注4] 开始读。

注 4： 事实上，正如之前从 SHOW MASTER STATUS 看到的，真正的日志起始位置是 98，一旦备库连接到主库就开始工作，现在连接还未发生。

运行下面的命令开始复制：

```
mysql> START SLAVE;
```

执行该命令没有显示错误，现在我们再用 SHOW SLAVE STATUS 命令检查：

```
mysql> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
             Slave_IO_State: Waiting for master to send event
                Master_Host: server1
                Master_User: repl
                Master_Port: 3306
              Connect_Retry: 60
            Master_Log_File: mysql-bin.000001
        Read_Master_Log_Pos: 164
             Relay_Log_File: mysql-relay-bin.000001
              Relay_Log_Pos: 164
      Relay_Master_Log_File: mysql-bin.000001
           Slave_IO_Running: Yes
          Slave_SQL_Running: Yes
                             ...omitted...
      Seconds_Behind_Master: 0
```

从输出可以看出 I/O 线程和 SQL 线程都已经开始运行，Seconds_Behind_Master 的值也不再为 NULL（稍后再解释 Seconds_Behind_Master 的含义）。I/O 线程正在等待从主库传递过来的事件，这意味着 I/O 线程已经读取了主库所有的事件。日志位置发生了变化，

455 表明已经从主库获取和执行了一些事件（你的结果可能会有所不同）。如果在主库上做一些数据更新，就会看到备库的文件或者日志位置都可能会增加。备库中的数据同样会随之更新。

我们还可以从线程列表中看到复制线程。在主库上可以看到由备库 I/O 线程向主库发起的连接。

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 55
   User: repl
   Host: replica1.webcluster_1:54813
     db: NULL
Command: Binlog Dump
   Time: 610237
  State: Has sent all binlog to slave; waiting for binlog to be updated
   Info: NULL
```

同样，在备库也可以看到两个线程，一个是 I/O 线程，一个是 SQL 线程：

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 1
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 611116
  State: Waiting for master to send event
   Info: NULL
*************************** 2. row ***************************
     Id: 2
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 33
  State: Has read all relay log; waiting for the slave I/O thread to update it
   Info: NULL
```

这些简单的输出来自一台已经运行了一段时间的服务器，所以 I/O 线程在主库和备库上的 `Time` 列的值较大。SQL 线程在备库已经空闲了 33 秒。这意味着 33 秒内没有重放任何事件。

这些线程总是运行在“system user”账号下，其他列的值则不相同。例如，当 SQL 线程回放事件时，`Info` 列可能显示正在执行的查询。

> 如果只是想实验 MySQL 的复制，Giuseppe Maxia 的 MySQL 沙箱脚本（*http://mysqlsandbox.net*）能够帮助你从一个之前下载的安装包中一次性安装。通过如下命令只需要几次按键和大约 15 秒，就可以运行一个主库和两个备库： 456
>
> ```
> $ ./set_replication.pl /path/to/mysql-tarball.tar.gz
> ```

10.2.4 从另一个服务器开始复制

前面的设置都是假定主备库均为刚刚安装好且都是默认的数据，也就是说两台服务器上数据相同，并且知道当前主库的二进制日志。这不是典型的案例。大多数情况下有一个已经运行了一段时间的主库，然后用一台新安装的备库与之同步，此时这台备库还没有数据。

有几种办法来初始化备库或者从其他服务器克隆数据到备库。包括从主库复制数据、从另外一台备库克隆数据，以及使用最近的一次备份来启动备库，需要有三个条件来让主库和备库保持同步：

- 在某个时间点的主库的数据快照。

- 主库当前的二进制日志文件，和获得数据快照时在该二进制日志文件中的偏移量，我们把这两个值称为日志文件坐标（*log file coordinates*）。通过这两个值可以确定二进制日志的位置。可以通过 SHOW MASTER STATUS 命令来获取这些值。
- 从快照时间到现在的二进制日志。

下面是一些从别的服务器克隆备库的方法：

使用冷备份

最基本的方法是关闭主库，把数据复制到备库（高效复制文件的方法参考附录 C）。重启主库后，会使用一个新的二进制日志文件，我们在备库通过执行 CHANGE MASTER TO 指向这个文件的起始处。这个方法的缺点很明显：在复制数据时需要关闭主库。

使用热备份

如果仅使用了 MyISAM 表，可以在主库运行时使用 *mysqlhotcopy* 或 *rsync* 来复制数据，更多细节参阅第 15 章。

使用 *mysqldump*

如果只包含 InnoDB 表，那么可以使用以下命令来转储主库数据并将其加载到备库，然后设置相应的二进制日志坐标：

457

```
$ mysqldump --single-transaction --all-databases --master-data=1--host=server1 \
  | mysql --host=server2
```

选项 *--single-transaction* 使得转储的数据为事务开始前的数据。如果使用的是非事务型表，可以使用 *--lock-all-tables* 选项来获得所有表的一致性转储。

使用快照或备份

只要知道对应的二进制日志坐标，就可以使用主库的快照或者备份来初始化备库（如果使用备份，需要确保从备份的时间点开始的主库二进制日志都要存在）。只需要把备份或快照恢复到备库，然后使用 CHANGE MASTER TO 指定二进制日志的坐标。第 15 章会介绍更多的细节，也可以使用 LVM 快照、SAN 快照、EBS 快照——任何快照都可以。

使用 *Percona Xtrabackup*

Percona 的 Xtrabackup 是一款开源的热备份工具，多年前我们就介绍过。它能够在备份时不阻塞服务器的操作，因此可以在不影响主库的情况下设置备库。可以通过克隆主库或另一个已存在的备库的方式来建立备库。

在 15 章会介绍更多使用 Percona Xtrabackup 的细节。这里会介绍一些相关的功能。创建一个备份（不管是从主库还是从别的备库），并将其转储到目标机器，然后根据备份获得正确的开始复制的位置。

- 如果是从主库获得备份，可以从 xtrabackup_binlog_pos_innodb 文件中获得复制开始的位置。
- 如果是从另外的备库获得备份，可以从 xtrabackup_slave_info 文件中获得复制开始的位置。

另外，在第 15 章提到的 InnoDB 热备份和 MySQL 企业版的备份，也是比较好的初始化备库方式。

使用另外的备库

可以使用任何一种提及的克隆或者拷贝技术来从任意一台备库上将数据克隆到另外一台服务器。但是如果使用的是 *mysqldump*，*--master-data* 选项就会不起作用。

此外，不能使用 SHOW MASTER STATUS 来获得主库的二进制日志坐标，而是在获取快照时使用 SHOW SLAVE STATUS 来获取备库在主库上的执行位置。

使用另外的备库进行数据克隆最大的缺点是，如果这台备库的数据已经和主库不同步，克隆得到的就是脏数据。

> 不要使用 LOAD DATA FROM MASTER 或者 LOAD TABLE FROM MASTER！这些命令过时、缓慢，并且非常危险，并且只适用于 MyISAM 存储引擎。

不管选择哪种技术，都要能熟练运用，要记录详细的文档或编写脚本。因为可能不止一 458
次需要做这样的事情。甚至当错误发生时，也需要能够处理。

10.2.5 推荐的复制配置

有许多参数来控制复制，其中一些会对数据安全和性能产生影响。稍后我们会解释何种规则在何时会失效。本小节推荐的一种“安全”的配置，可以最小化问题发生的概率。

在主库上二进制日志最重要的选项是 sync_binlog：

```
sync_binlog=1
```

如果开启该选项，MySQL 每次在提交事务前会将二进制日志同步到磁盘上，保证在服务器崩溃时不会丢失事件。如果禁止该选项，服务器会少做一些工作，但二进制日志文件可能在服务器崩溃时损坏或丢失信息。在一个不需要作为主库的备库上，该选项带来了不必要的开销。它只适用于二进制日志，而非中继日志。

如果无法容忍服务器崩溃导致表损坏，推荐使用 InnoDB。在表损坏无关紧要时，MyISAM 是可以接受的，但在一次备库服务器崩溃重启后，MyISAM 表可能已经处于不一致状态。一种可能是语句没有完全应用到一个或多个表上，那么即使修复了表，数据也可能是不一致的。

如果使用 InnoDB，我们强烈推荐设置如下选项：

```
innodb_flush_logs_at_trx_commit   # Flush every log write
innodb_support_xa=1               # MySQL 5.0 and newer only
innodb_safe_binlog                # MySQL 4.1 only, roughly equivalent to
                                  # innodb_support_xa
```

这些是 MySQL 5.0 及最新版本中的默认配置，我们推荐明确指定二进制日志的名字，以保证二进制日志名在所有服务器上是一致的，避免因为服务器名的变化导致的日志文件名变化。你可能认为以服务器名来命名二进制日志无关紧要，但经验表明，当在服务器间转移文件、克隆新的备库、转储备份或者其他一些你想象不到的场景下，可能会导致很多问题。为了避免这些问题，需要给 log_bin 选项指定一个参数。可以随意地给一个绝对路径，但必须明确地指定基本的命名（正如本章之前讨论的）。

```
log_bin=/var/lib/mysql/mysql-bin  # Good; specifies a path and base name
#log_bin                          # Bad; base name will be server's hostname
```

459 在备库上，我们同样推荐开启如下配置选项，为中继日志指定绝对路径：

```
relay_log=/path/to/logs/relay-bin
skip_slave_start
read_only
```

通过设置 relay_log 可以避免中继日志文件基于机器名来命名，防止之前提到的可能在主库发生的问题。指定绝对路径可以避免多个 MySQL 版本中存在的 Bug，这些 Bug 可能会导致中继日志在一个意料外的位置创建。skip_slave_start 选项能够阻止备库在崩溃后自动启动复制。这可以给你一些机会来修复可能发生的问题。如果备库在崩溃后自动启动并且处于不一致的状态，就可能会导致更多的损坏，最后将不得不把所有数据丢弃，并重新开始配置备库。

read_only 选项可以阻止大部分用户更改非临时表，除了复制 SQL 线程和其他拥有超级权限的用户之外，这也是要尽量避免给正常账号授予超级权限的原因之一。

即使开启了所有我们建议的选项，备库仍然可能在崩溃后被中断，因为 *master.info* 和中继日志文件都不是崩溃安全的。默认情况下甚至不会刷新到磁盘，直到 MySQL 5.5 版本才有选项来控制这种行为。如果正在使用 MySQL 5.5 并且不介意额外的 fsync() 导致的性能开销，最好设置以下选项：

```
sync_master_info      = 1
sync_relay_log        = 1
sync_relay_log_info   = 1
```

如果备库与主库的延迟很大，备库的 I/O 线程可能会写很多中继日志文件，SQL 线程在重放完一个中继日志中的事件后会尽快将其删除（通过 relay_log_purge 选项来控制）。

但如果延迟非常严重，I/O 线程可能会把整个磁盘撑满。解决办法是配置 relay_log_space_limit 变量。如果所有中继日志的大小之和超过这个值，I/O 线程会停止，等待 SQL 线程释放磁盘空间。

尽管听起来很美好，但有一个隐藏的问题。如果备库没有从主库上获取所有的中继日志，这些日志可能在主库崩溃时丢失。早先这个选项存在一些 Bug，使用率也不高，所以用到这个选项遇到 Bug 的风险会更高。除非磁盘空间真的非常紧张，否则最好让中继日志使用其需要的磁盘空间，这也是为什么我们没有将 relay_log_space_limit 列入推荐的配置选项的原因。

10.3 复制的原理

460

我们已经介绍了复制的一些基本概念，接下来要更深入地了解复制。让我们看看复制究竟是如何工作的，有哪些优点和弱点，最后介绍一些更高级的复制配置选项。

10.3.1 基于语句的复制

在 MySQL 5.0 及之前的版本中只支持基于语句的复制（也称为逻辑复制），这在数据库领域是很少见的。基于语句的复制模式下，主库会记录那些造成数据更改的查询，当备库读取并重放这些事件时，实际上只是把主库上执行过的 SQL 再执行一遍。这种方式既有好处，也有缺点。

最明显的好处是实现相当简单。理论上讲，简单地记录和执行这些语句，能够让主备保持同步。另一个好处是二进制日志里的事件更加紧凑，所以相对而言，基于语句的模式不会使用太多带宽。一条更新好几兆数据的语句在二进制日志里可能只占几十个字节。另外 *mysqlbinlog* 工具（本章多处会提到）是使用基于语句的日志的最佳工具。

但事实上基于语句的方式可能并不如其看起来那么便利。因为主库上的数据更新除了执行的语句外，可能还依赖于其他因素。例如，同一条 SQL 在主库和备库上执行的时间可能稍微或很不相同，因此在传输的二进制日志中，除了查询语句，还包括了一些元数据信息，如当前的时间戳。即便如此，还存在着一些无法被正确复制的 SQL。例如，使用 CURRENT_USER() 函数的语句。存储过程和触发器在使用基于语句的复制模式时也可能存在问题。

另外一个问题是更新必须是串行的。这需要更多的锁——有时候要特别关注这一点。另外不是所有的存储引擎都支持这种复制模式。尽管这些存储引擎是包括在 MySQL 5.5 及之前版本中发行的。

可以在 MySQL 手册与复制相关的章节中找到基于语句的复制存在的限制的完整列表。

10.3.2 基于行的复制

MySQL 5.1 开始支持基于行的复制，这种方式会将实际数据记录在二进制日志中，跟其他数据库的实现比较相像。它有其自身的一些优点和缺点。最大的好处是可以正确地复制每一行。一些语句可以被更加有效地复制。

461 基于行的复制没有向后兼容性，和 MySQL 5.1 一起发布的 *mysqlbinlog* 工具可以读取基于行的复制的事件格式（它对人是不可读的，但 MySQL 可以解释），但是早期版本的 *mysqlbinlog* 无法识别这类事件，在遇到错误时会退出。

由于无须重放更新主库数据的查询，使用基于行的复制模式能够更高效地复制数据。重放一些查询的代价可能会很高。例如，下面有一个查询将数据从一个大表中汇总到小表：

```
mysql> INSERT INTO summary_table(col1, col2, sum_col3)
    -> SELECT col1, col2, sum(col3)
    -> FROM enormous_table
    -> GROUP BY col1, col2;
```

想象一下，如果表 enormous_table 的列 col1 和 col2 有三种组合，这个查询可能在源表上扫描多次，但最终只在目标表上产生三行数据。但使用基于行的复制方式，在备库上开销会小很多。这种情况下，基于行的复制模式更加高效。

但在另一方面，下面这条语句使用基于语句的复制方式代价会小很多：

```
mysql> UPDATE enormous_table SET col1 = 0;
```

由于这条语句做了全表更新，使用基于行的复制开销会很大，因为每一行的数据都会被记录到二进制日志中，这使得二进制日志事件非常庞大。并且会给主库上记录日志和复制增加额外的负载，更慢的日志记录则会降低并发度。

由于没有哪种模式对所有情况都是完美的，MySQL 能够在这两种复制模式间动态切换。默认情况下使用的是基于语句的复制方式，但如果发现语句无法被正确地复制，就切换到基于行的复制模式。还可以根据需要来设置会话级别的变量 binlog_format，控制二进制日志格式。

对于基于行的复制模式，很难进行时间点恢复，但这并非不可能。稍后讲到的日志服务器对此会有帮助。

10.3.3 基于行或基于语句：哪种更优

我们已经讨论了这两种复制模式的优点和缺点，那么在实际应用中哪种方式更优呢？

理论上基于行的复制模式整体上更优，并且在实际应用中也适用于大多数场景。但这种方式太新了以至于没有将一些特殊的功能加入到其中来满足数据库管理员的操作需求。因此一些人直到现在还没有开始使用。以下详细地阐述两种方式的优点和缺点，以帮助你决定哪种方式更合适。 462

基于语句的复制模式的优点

当主备的模式不同时，逻辑复制能够在多种情况下工作。例如，在主备上的表的定义不同但数据类型相兼容、列的顺序不同等情况。这样就很容易先在备库上修改 schema，然后将其提升为主库，减少停机时间。基于语句的复制方式一般允许更灵活的操作。

基于语句的方式执行复制的过程基本上就是执行 SQL 语句。这意味着所有在服务器上发生的变更都以一种容易理解的方式运行。这样当出现问题时可以很好地去定位。

基于语句的复制模式的缺点

很多情况下通过基于语句的模式无法正确复制，几乎每一个安装的备库都会至少碰到一次。事实上对于存储过程，触发器以及其他的一些语句的复制在 5.0 和 5.1 的一系列版本中存在大量的 Bug。这些语句的复制的方式已经被修改了很多次，以使其更好地工作。简单地说：如果正在使用触发器或者存储过程，就不要使用基于语句的复制模式，除非能够清楚地确定不会碰到复制问题。

基于行的复制模式的优点

几乎没有基于行的复制模式无法处理的场景。对于所有的 SQL 构造、触发器、存储过程等都能正确执行。只是当你试图做一些诸如在备库修改表的 schema 这样的事情时才可能导致复制失败。

这种方式同样可能减少锁的使用，因为它并不要求这种强串行化是可重复的。

基于行的复制模式会记录数据变更，因此在二进制日志中记录的都是实际上在主库上发生了变化的数据。你不需要查看一条语句去猜测它到底修改了哪些数据。在某种程度上，该模式能够更加清楚地知道服务器上发生了哪些更改，并且有一个更好的数据变更记录。另外在一些情况下基于行的二进制日志还会记录发生改变之前的数据，因此这可能有利于某些数据恢复。 463

在很多情况下，由于无须像基于语句的复制那样需要为查询建立执行计划并执行查询，因此基于行的复制占用更少的 CPU。

最后，在某些情况下，基于行的复制能够帮助更快地找到并解决数据不一致的情况。举个例子，如果是使用基于语句的复制模式，在备库更新一个不存在的记录时不会失败，但在基于行的复制模式下则会报错并停止复制。

基于行的复制模式的缺点

由于语句并没有在日志里记录，因此无法判断执行了哪些 SQL，除了需要知道行的变化外，这在很多情况下也很重要（这可能在未来的 MySQL 版本中被修复）。

使用一种完全不同的方式在备库进行数据变更——而不是执行 SQL。事实上，执行基于行的变化的过程就像一个黑盒子，你无法知道服务器正在做什么。并且没有很好的文档和解释。因此当出现问题时，可能很难找到问题所在。例如，若备库使用一个效率低下的方式去寻找行记录并更新，你无法观察到这一点。

如果有多层的复制服务器，并且所有的都被配置成基于行的复制模式，当会话级别的变量 `@@binlog_format` 被设置成 `STATEMENT` 时，所执行的语句在源服务器上被记录为基于语句的模式，但第一层的备库可能将其记录成行模式，并传递给其他层的备库。也就是说你期望的基于语句的日志在复制拓扑中将会被切换到基于行的模式。

基于行的日志无法处理诸如在备库修改表的 schema 这样的情况，而基于语句的日志可以。

在某些情况下，例如找不到要修改的行时，基于行的复制可能会导致复制停止，而基于语句的复制则不会。这也可以认为是基于行的复制的一个优点。该行为可以通过 `slave_exec_mode` 来进行配置。

这些缺点正在被慢慢解决，但直到写作本书时，它们在大多数生产环境中依然存在。

10.3.4 复制文件

让我们来看看复制会使用到的一些文件。前面已经介绍了二进制日志文件和中继日志文件，其实还有其他的文件会被用到。不同版本的 MySQL 默认情况下可能将这些文件放
464 到不同的目录里，大多取决具体的配置选项。可能在 data 目录或者包含服务器 *.pid* 文件的目录下（对于类 UNIX 系统可能是 */var/run/mysqld*）。它们的详细介绍如下。

mysql-bin.index

当在服务器上开启二进制日志时，同时会生成一个和二进制日志同名的但以 *.index* 作为后缀的文件，该文件用于记录磁盘上的二进制日志文件。这里的“index”并不是指表的索引，而是说这个文件的每一行包含了二进制文件的文件名。

你可能认为这个文件是多余的，可以被删除（毕竟 MySQL 可以在磁盘上找到它需要的文件）。事实上并非如此，MySQL 依赖于这个文件，除非在这个文件里有记录，否则 MySQL 识别不了二进制日志文件。

mysql-relay-bin-index

这个文件是中继日志的索引文件，和 *mysql-bin.index* 的作用类似。

master.info

这个文件用于保存备库连接到主库所需要的信息，格式为纯文本（每行一个值），不同的 MySQL 版本，其记录的信息也可能不同。此文件不能删除，否则备库在重启后无法连接到主库。这个文件以文本的方式记录了复制用户的密码，所以要注意此文件的权限控制。

relay-log.info

这个文件包含了当前备库复制的二进制日志和中继日志坐标（例如，备库复制在主库上的位置），同样也不要删除这个文件，否则在备库重启后将无法获知从哪个位置开始复制，可能会导致重放已经执行过的语句。

使用这些文件来记录 MySQL 复制和日志状态是一种非常粗糙的方式。更不幸的是，它们不是同步写的。如果服务器断电并且文件数据没有被刷新到磁盘，在重启服务器后，文件中记录的数据可能是错误的。正如之前提到的，这些问题在 MySQL 5.5 里做了改进。

以 *.index* 作为后缀的文件也与设置 expire_logs_days 存在交互，该参数定义了 MySQL 清理过期日志的方式，如果文件 *mysql-bin.index* 在磁盘上不存在，在某些 MySQL 版本自动清理就会不起作用，甚至执行 PURGE MASTER LOGS 语句也没有用。这个问题的解决方法通常是使用 MySQL 服务器管理二进制日志，这样就不会产生误解（这意味着不应该使用 *rm* 来自己清理日志）

最好能显式地执行一些日志清理策略，比如设置 expire_logs_days 参数或者其他方式，否则 MySQL 的二进制日志可能会将磁盘撑满。当做这些事情时，还需要考虑到备份策略。

10.3.5 发送复制事件到其他备库

465

log_slave_updates 选项可以让备库变成其他服务器的主库。在设置该选项后，MySQL 会将其执行过的事件记录到它自己的二进制日志中。这样它的备库就可以从其日志中检索并执行事件。图 10-2 阐述了这一过程。

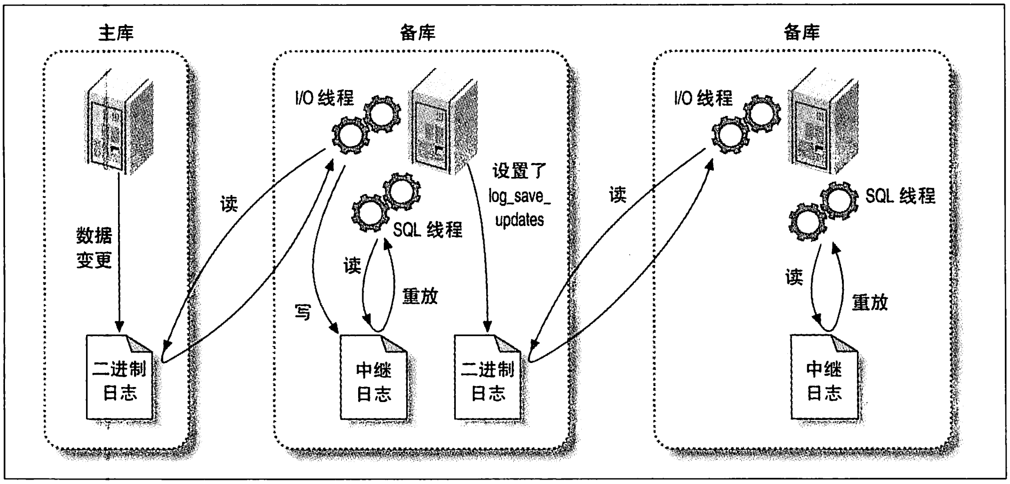


图10-2：将复制事件传递到更多的备库

在这种场景下，主库将数据更新事件写入二进制日志，第一个备库提取并执行这个事件。这时候一个事件的生命周期应该已经结束了，但由于设置了 `log_slave_updates`，备库会将这个事件写到它自己的二进制日志中。这样第二个备库就可以将事件提取到它的中继日志中并执行。这意味着作为源服务器的主库可以将其数据变化传递给没有与其直接相连的备库上。默认情况下这个选项是被打开的，这样在连接到备库时就不需要重启服务器。

当第一个备库将从主库获得的事件写入到其二进制日志中时，这个事件在备库二进制日志中的位置与其在主库二进制日志中的位置几乎肯定是不相同的，可能在不同的日志文件或文件内不同的位置。这意味着你不能假定所有拥有同一逻辑复制点的服务器拥有相同的日志坐标。稍后我们会提到，这种情况会使某些任务更加复杂，例如，修改一个备库的主库或将备库提升为主库。

除非你已经注意到要给每个服务器分配一个唯一的服务器 ID，否则按照这种方式配置备库会导致一些奇怪的错误，甚至还会导致复制停止。一个更常见的问题是：为什么
466 要指定服务器 ID，难道 MySQL 在不知道复制命令来源的情况下不能执行吗？为什么 MySQL 要在意服务器 ID 是全局唯一的。问题的答案在于 MySQL 在复制过程中如何防止无限循环。当复制 SQL 线程读中继日志时，会丢弃事件中记录的服务器 ID 和该服务器本身 ID 相同的事件，从而打破了复制过程中的无限循环。在某些复制拓扑结构下打破无限循环非常重要，例如主 - 主复制结构[注5]。

> 如果在设置复制的时候碰到问题，服务器 ID 应该是需要检查的因素之一。当然只检查 `@@server_id` 是不够的，它有一个默认值，除非在 *my.cnf* 文件或通过 SET 命令明确指定它的值，复制才会工作。如果使用 SET 命令，确保同时也更新了配置文件，否则 SET 命令的设定可能在服务器重启后丢失。

10.3.6 复制过滤器

复制过滤选项允许你仅复制服务器上一部分数据，不过这可能没有想象中那么好用。有两种复制过滤方式：在主库上过滤记录到二进制日志中的事件，以及在备库上过滤记录到中继日志的事件。图 10-3 显示了这两种类型。

注 5：语句在无限循环中来回传递也是多服务器环形复制拓扑结构中比较有意思的话题之一，后面我们会提到。要尽量避免环形复制。

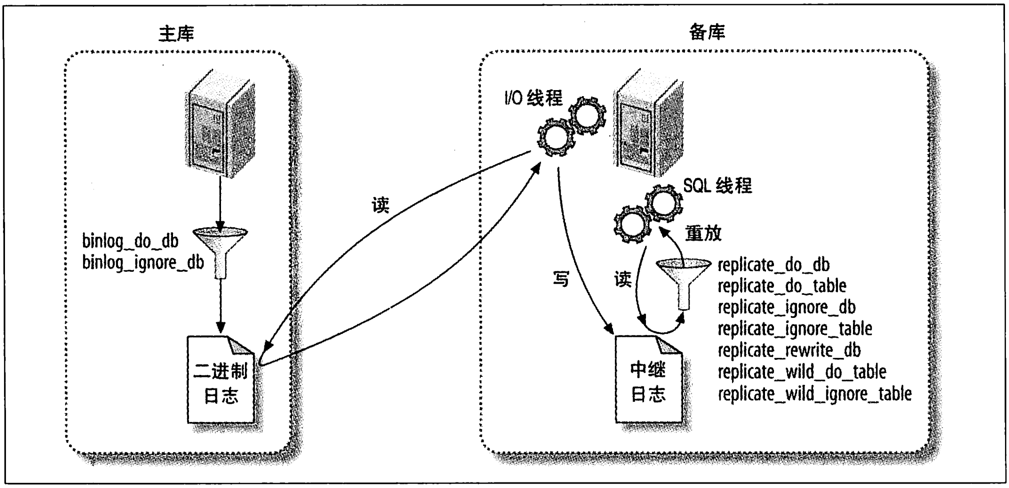


图10-3：复制过滤选项

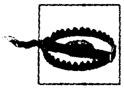
使用选项 binlog_do_db 和 binlog_ignore_db 来控制过滤，稍后我们会解释为什么通常不需要开启它们，除非你乐于向老板解释为什么数据会永久丢失并且无法恢复。

在备库上，可以通过设置 replicate_* 选项，在从中继日志中读取事件时进行过滤。你可以复制或忽略一个或多个数据库，把一个数据库重写到另外一个数据库，或使用类似 LIKE 的模式复制或忽略数据库表。 467

要理解这些选项，最重要是弄清楚 *_do_db 和 *_ignore_db 在主库和备库上的意义，它们可能不会按照你所设想的那样工作。你可能会认为它会根据目标数据库名过滤，但实际上过滤的是当前的默认数据库[注6]。也就是说，如果在主库上执行如下语句：

```
mysql> USE test;
mysql> DELETE FROM sakila.film;
```

*_do_db 和 *_ignore_db 都会在数据库 test 上过滤 DELETE 语句，而不是在 sakila 上。这通常不是想要的结果，可能会导致执行或忽略错误的语句。*_do_db 和 *_ignore_db 有一些作用，但非常有限。必须要很小心地去使用这些参数，否则很容易造成主备不同步或复制出错。



binlog_do_db 和 binlog_ignore_db 不仅可能会破坏复制，还可能会导致从某个时间点的备份进行数据恢复时失败。在大多数情况下都不应该使用这些参数。本章稍后部分我们展示了一些使用 blackhole 表进行复制过滤的方法。

注6： 如果使用的是基于语句的复制，就会有这样的问题，但基于行的复制方式则不会（另一个远离它们的理由）。

总地来说，复制过滤随时可能会发生问题。举个例子，假如要阻止赋权限操作传递给备库，这种需求是很普遍的。（提醒一下，这样做可能是错误的，有别的更好的方式来达成真正的目的）。过滤系统表的复制当然能够阻止 GRANT 语句的复制，但同样也会阻止事件和定时任务的复制。正是这些不可预知的后果，使用复制过滤要非常慎重。更好的办法是阻止一些特殊的语句被复制，通常是设置 SQL_LOG_BIN=0，虽然这种方法也有它的缺点。总地来说，除非万不得已，不要使用复制过滤，因为它很容易中断复制并导致问题，在需要灾难恢复时也会带来极大的不方便。

过滤选项在 MySQL 文档里介绍得很详细，因此本书不再重复更多的细节。

468

10.4 复制拓扑

可以在任意个主库和备库之间建立复制，只有一个限制：每一个备库只能有一个主库。有很多复杂的拓扑结构，但即使是最简单的也可能会非常灵活。一种拓扑可以有多种用途。关于使用复制的不同方式可以很轻易地写一本书。

我们已经讨论了如何为主库设置一个备库，本节我们讨论其他比较普遍的拓扑结构以及它们的优缺点。记住下面的基本原则：

- 一个 MySQL 备库实例只能有一个主库。
- 每个备库必须有一个唯一的服务器 ID。
- 一个主库可以有多个备库（或者相应的，一个备库可以有多个兄弟备库）。
- 如果打开了 log_slave_updates 选项，一个备库可以把其主库上的数据变化传播到其他备库。

10.4.1 一主库多备库

除了我们已经提过的两台服务器的主备结构外，这是最简单的拓扑结构。事实上一主多备的结构和基本配置差不多简单，因为备库之间根本没有交互[注7]，它们仅仅是连接到同一个主库上。图 10-4 显示了这种结构。

469 在有少量写和大量读时，这种配置是非常有用的。可以把读分摊到多个备库上，直到备库给主库造成了太大的负担，或者主备之间的带宽成为瓶颈为止。你可以按照之前介绍的方法一次性设置多个备库，或者根据需要增加备库。

注 7： 从技术上讲这并非正确的。但如果有重复的服务器 ID，它们将陷入竞争，并反复将对方从主库上踢出。

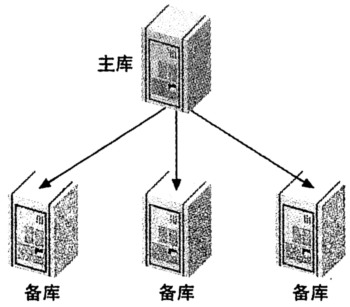


图10-4：一主多备结构

尽管这是非常简单的拓扑结构，但它非常灵活，能满足多种需求。下面是它的一些用途：

- 为不同的角色使用不同的备库（例如添加不同的索引或使用不同的存储引擎）。
- 把一台备库当作待用的主库，除了复制没有其他数据传输。
- 将一台备库放到远程数据中心，用作灾难恢复。
- 延迟一个或多个备库，以备灾难恢复。
- 使用其中一个备库，作为备份、培训、开发或者测试使用服务器。

这种结构流行的原因是它避免了很多其他拓扑结构的复杂性。例如：可以方便地比较不同备库重放的事件在主库二进制日志中的位置。换句话说，如果在同一个逻辑点停止所有备库的复制，它们正在读取的是主库上同一个日志文件的相同物理位置。这是个很好的特性，可以减轻管理员许多工作，例如把备库提升为主库。

这种特性只存在于兄弟备库之间。在没有直接的主备或者兄弟关系的服务器上去比较日志文件的位置要复杂很多。之后我们会提到的许多拓扑结构，例如树形复制或分布式主库，很难计算出复制的事件的逻辑顺序。

10.4.2 主动 - 主动模式下的主 - 主复制

主 - 主复制（也叫双主复制或双向复制）包含两台服务器，每一个都被配置成对方的主库和备库，换句话说，它们是一对主库。图 10-5 显示了该结构。

图10-5：主–主复制

主动－主动模式下主－主复制有一些应用场景，但通常用于特殊的目的。一个可能的应用场景是两个处于不同地理位置的办公室，并且都需要一份可写的数据拷贝。

这种配置最大的问题是如何解决冲突，两个可写的互主服务器导致的问题非常多。这通常发生在两台服务器同时修改一行记录，或同时在两台服务器上向一个包含 AUTO_INCREMENT 列的表里插入数据[注8]。

470

MySQL 不支持多主库复制

多主库复制（multisource replication）特指一个备库有多个主库。不管之前你知道什么，但 MySQL（和其他数据库产品不一样）现在不支持如图 10-6 所示的结构，本章稍后我们会向你介绍如何模仿多主库复制。

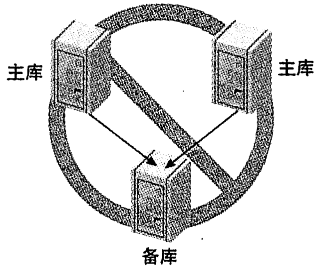


图10-6：MySQL不支持多主库复制

MySQL 5.0 增加了一些特性，使得这种配置稍微安全了点，就是设置 auto_increment_increment 和 auto_increment_offset。通过这两个选项可以让 MySQL 自动为 INSERT 语句选择不互相冲突的值。然而允许向两台主库上写入仍然很危险。在两台机器上根据不同的顺序更新，可能会导致数据不同步。例如，一个只有一列的表，只有一行值为 1 的记录，假设同时执行下面两条语句：

471

- 在第一台主库上：

```
mysql> UPDATE tbl SET col=col + 1;
```

- 在第二台主库上：

```
mysql> UPDATE tbl SET col=col * 2;
```

注 8： 事实上这些问题经常一周发生三次，并且我们也发现需要好几个月才能解决这些问题。

那么结果呢？一台服务器上值为 4，另一台的值为 3，并且没有报告任何复制错误。

数据不同步还仅仅是开始。当正常的复制发生错误停止了，但应用仍在同时向两台服务器写入数据，这时候会发生什么呢？你不能简单地把数据从一台服务器复制到另外一台，因为这两台机器上需要复制的数据都可能发生了变化。解决这个问题将会非常困难。

如果足够仔细地配置这种架构，例如很好地划分数据和权限，并且你很清楚自己在做什么，可以避免一些问题[注9]。然而这通常很难做好，并且有更好的办法来实现你所需要的。

总地来说，允许向两个服务器上写入所带来的麻烦远远大于其带来的好处，但下一节描述的主动 - 被动模式则会非常有用。

10.4.3 主动 - 被动模式下的主 - 主复制

这是前面描述的主 - 主结构的变体，它能够避免我们之前讨论的问题。这也是构建容错性和高可用性系统的非常强大的方式，主要区别在于其中的一台服务器是只读的被动服务器，如图 10-7 所示。

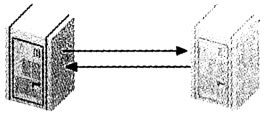


图10-7：主动-被动模式下的主-主复制

这种方式使得反复切换主动和被动服务器非常方便，因为服务器的配置是对称的。这使得故障转移和故障恢复很容易。它也可以让你在不关闭服务器的情况下执行维护、优化表、升级操作系统（或者应用程序、硬件等）或其他任务。

例如，执行 ALTER TABLE 操作可能会锁住整个表，阻塞对表的读和写，这可能会花费很长时间并导致服务中断。然而在主 - 主配置下，可以先停止主动服务器上的备库复制线程（这样就不会在被动服务器上执行任何更新），然后在被动服务器上执行 ALTER 操作，交换角色，最后在先前的主动服务器上[注10]启动复制线程。这个服务器将会读取中继日志并执行相同的 ALTER 语句。这可能花费很长时间，但不要紧，因为该服务器没有为任何活跃查询提供服务。 472

主动 - 被动模式的主 - 主结构能够帮助回避许多 MySQL 的问题和限制，此外还有一些

注 9： 一些，但不是全部——我们可以吹毛求疵，并指出任何你可以想象的漏洞。

注 10： 可以通过设置 SQL_LOG_BIN=0 来暂时禁止记录二进制日志而无须停止复制。一些语句，例如 Optimize TABLE，也支持 LOCAL 或者 NO_WRITE_TO_BINLOG 这些停止日志的选项。

工具可以完成这种类型的操作。

让我们看看如何配置主 - 主服务器对，在两台服务器上执行如下设置后，会使其拥有对称的设置：

1. 确保两台服务器上有相同的数据。
2. 启用二进制日志，选择唯一的服务器 ID，并创建复制账号。
3. 启用备库更新的日志记录，后面将会看到，这是故障转移和故障恢复的关键。
4. 把被动服务器配置成只读，防止可能与主动服务器上的更新产生冲突，这一点是可选的。
5. 启动每个服务器的 MySQL 实例。
6. 将每个主库设置为对方的备库，使用新创建的二进制日志开始工作。

让我们看看主动服务器上更新时会发生什么事情。更新被记录到二进制日志中，通过复制传递给被动服务器的中继日志中。被动服务器执行查询并将其记录到自己的二进制日志中(因为开启了 `log_slave_updates` 选项)。由于事件的服务器 ID 与主动服务器的相同，因此主动服务器将忽略这些事件。在后面的"修改主库"可了解更多的角色切换相关内容。

设置主动 - 被动的主 - 主拓扑结构在某种意义上类似于创建一个热备份，但是可以使用这个"备份"来提高性能，例如，用它来执行读操作、备份、"离线"维护以及升级等。真正的热备份做不了这些事情。然而，你不会获得比单台服务器更好的写性能（稍后会提到）。

当我们讨论使用复制的场景和用途时，还会提到这种复制方式。它是一种非常常见并且重要的拓扑结构。

473

10.4.4 拥有备库的主 - 主结构

另外一种相关的配置是为每个主库增加一个备库，如图 10-8 所示。

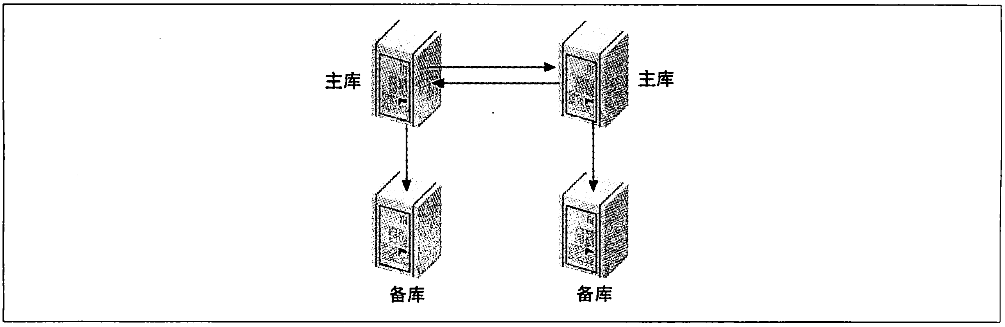


图10-8：拥有备库的主-主结构

这种配置的优点是增加了冗余，对于不同地理位置的复制拓扑，能够消除站点单点失效的问题。你也可以像平常一样，将读查询分配到备库上。

如果在本地为了故障转移使用主 - 主结构，这种配置同样有用。当主库失效时，用备库来代替主库还是可行的，虽然这有点复杂。同样也可以把备库指向一个不同的主库，但需要考虑增加的复杂度。

10.4.5 环形复制

如图 10-9 所示，双主结构实际上是环形结构的一种特例[注11]。环形结构可以有三个或更多的主库。每个服务器都是在它之前的服务器的备库，是在它之后的服务器的主库。这种结构也称为环形复制（*circular replication*）。

环形结构没有双主结构的一些优点，例如对称配置和简单的故障转移，并且完全依赖于环上的每一个可用节点，这大大增加了整个系统失效的几率。如果从环中移除一个节点，这个节点发起的事件就会陷入无限循环：它们将永远绕着服务器链循环。因为唯一可以根据服务器 ID 将其过滤的服务器是创建这个事件的服务器。总地来说，环形结构非常脆弱，应该尽量避免。

474

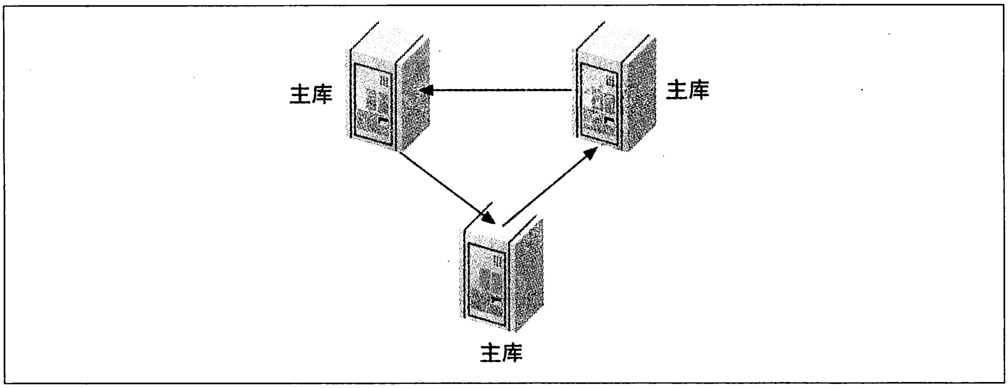


图10-9：环形复制拓扑

可以通过为每个节点增加备库的方式来减少环形复制的风险，如图 10-10 所示。但这仅仅防范了服务器失效的危险，断电或者其他一些影响到网络连接的问题都可能破坏整个环。

注 11： 也许应该说，是更明智的特例。

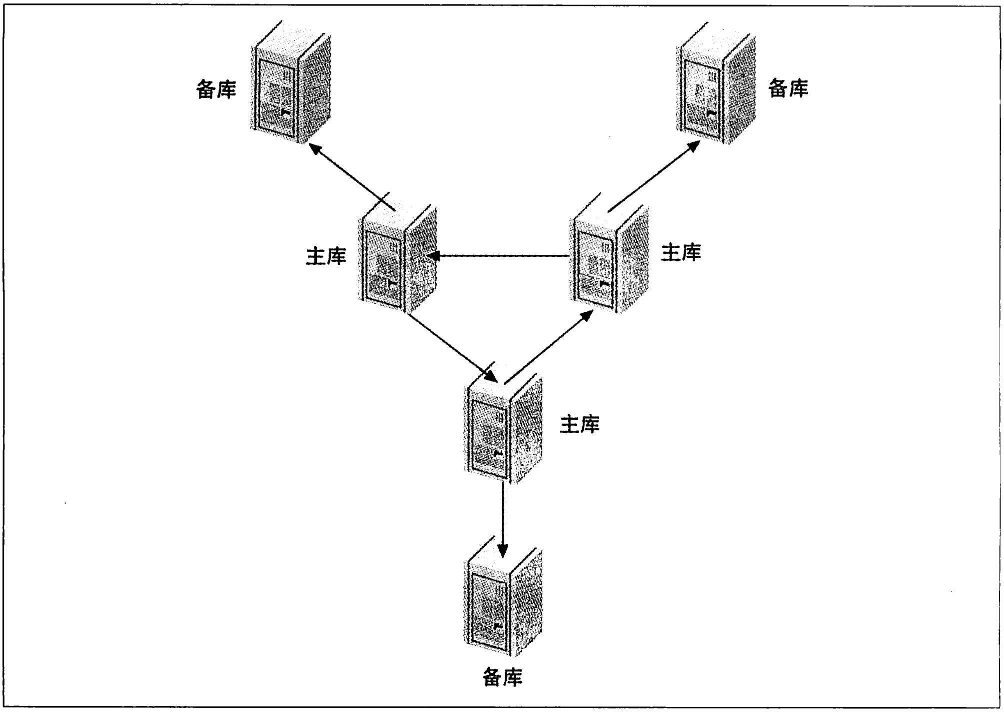


图10-10：拥有备库的环形结构

10.4.6 主库、分发主库以及备库

我们之前提到当备库足够多时，会对主库造成很大的负载。每个备库会在主库上创建一
475 个线程，并执行 *binlog dump* 命令。该命令会读取二进制日志文件中的数据并将其发送给备库。每个备库都会重复这样的工作，它们不会共享 binlog dump 的资源。

如果有很多备库，并且有大的事件时，例如一次很大的 LOAD DATA INFILE 操作，主库上的负载会显著上升，甚至可能由于备库同时请求同样的事件而耗尽内存并崩溃。另一方面，如果备库请求的数据不在文件系统的缓存中，可能会导致大量的磁盘检索，这同样会影响主库的性能并增加锁的竞争。

因此，如果需要多个备库，一个好办法是从主库移除负载并使用分发主库。分发主库事实上也是一个备库，它的唯一目的就是提取和提供主库的二进制日志。多个备库连接到分发主库，这使原来的主库摆脱了负担。为了避免在分发主库上做实际的查询，可以将它的表修改为 blackhole 存储引擎，如图 10-11 所示。

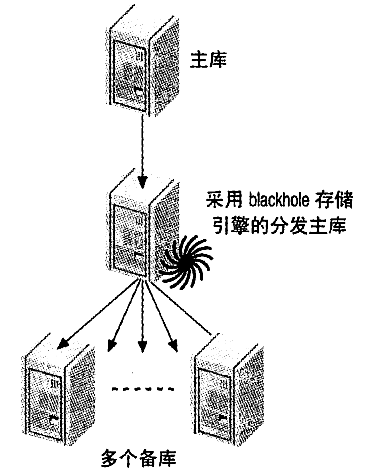


图10-11：一个主库、一个分发主库和多个备库

很难说当备库数据达到多少时需要一个分发主库。按照通用准则，如果主库接近满负载，不应该为其建立 10 个以上的备库。如果有少量的写操作，或者只复制其中一部分表，主库就可以提供更多的复制。另外，也不一定只使用一个分发主库。如果需要的话，可以使用多个分发主库向大量的备库进行复制，或者使用金字塔状的分发主库。在某些情况下，可以通过设置 slave_compressed_protocol 来节约一些主库带宽。这对跨数据中心复制很有好处。

还可以通过分发主库实现其他目的，例如，对二进制日志事件执行过滤和重写规则。这比在每个备库上重复进行日志记录、重写和过滤要高效得多。 476

如果在分发主库上使用 blackhole 表，可以支持更多的备库。虽然会在分发主库执行查询，但其代价非常小，因为 blackhole 表中没有任何数据。blockhole 表的缺点是其存在 Bug，例如在某些情况下会忘记将自增 ID 写入到二进制日志中。所以要小心使用 blackhole 表[注 12]。

一个比较常见的问题是如何确保分发服务器上的每个表都是 blackhole 存储引擎。如果有人在主库创建了一个表并指定了不同的存储引擎呢？确实，不管什么时候，在备库上使用不同的存储引擎总会导致同样的问题。常见的解决方案是设置服务器的 storage_engine 选项：

```
storage_engine = blackhole
```

注 12： 从 MySQL Bug 35178 和 62829 开始查阅，总地来说，如果使用的是不标准的存储引擎特性，最好去看看那些打开或者关闭的受影响的 Bug。

这只会影响那些没有指定存储引擎的CREATE TABLE的语句。如果有一个无法控制的应用，这种拓扑结构可能会非常脆弱。可以通过 skip_innodb 选项禁止 InnoDB，将表退化为 MyISAM。但你无法禁止 MyISAM 或者 Memory 引擎。

使用分发主库另外一个主要的缺点是无法使用一个备库来代替主库。因为由于分发主库的存在，导致各个备库与原始主库的二进制日志坐标已经不相同[注13]。

10.4.7 树或金字塔形

如果正在将主库复制到大量的备库中。不管是把数据分发到不同的地方，还是提供更高的读性能，使用金字塔结构都能够更好地管理，如图 10-12 所示。

这种设计的好处是减轻了主库的负担，就像前一节提到的分发主库一样。它的缺点是中间层出现的任何错误都会影响到多个服务器。如果每个备库和主库直接相连就不会存在这样的问题。同样，中间层次越多，处理故障会更困难、更复杂。

477

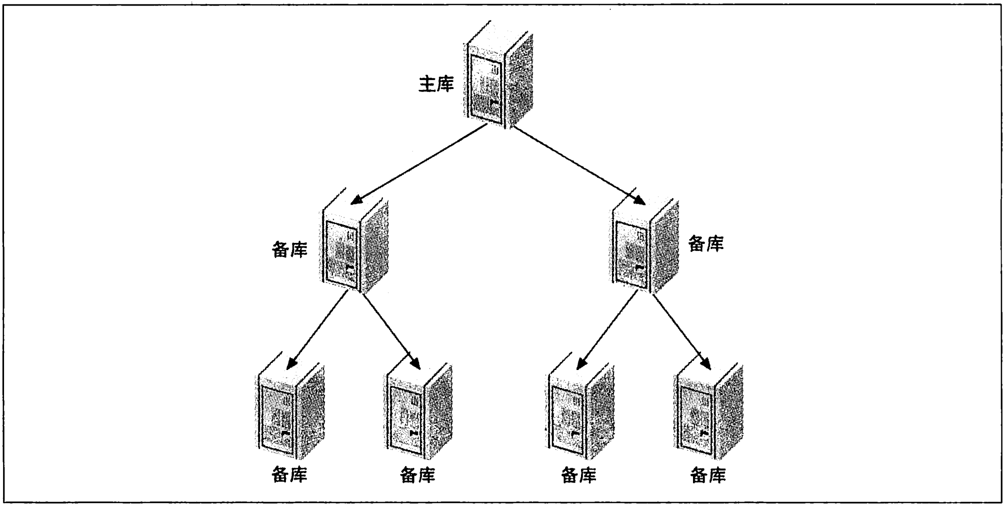


图10-12：金字塔形复制拓扑

10.4.8 定制的复制方案

MySQL 的复制非常灵活，可以根据需要定制解决方案。典型的定制方案包括组合过滤、分发和向不同的存储引擎复制。也可以使用“黑客手段”，例如，从一个使用 blackhole 存储引擎的服务器上复制或复制到这样的服务器上（本章已讨论过）。可以根据需要任

注 13： 可以使用 Percona 的工具集中的 *pt-heartbeat* 来创建一个粗糙的全局事务 ID。这样可以很方便地在多个服务器上寻找二进制日志的位置。因为“心跳表”本身就记录了大概的二进制日志位置。

意设计。这其中最大的限制是合理地监控和管理，以及所拥有资源的约束（网络带宽、CPU 能力等）。

选择性复制

为了利用访问局部性原理（locality of reference），并将需要读的工作集驻留在内存中，可以复制少量数据到备库中。如果每个备库只拥有主库的一部分数据，并且将读分配给备库，就可以更好地利用备库的内存。并且每个备库也只有主库一部分的写入负载，这样主库的能力更强并能保证备库延迟。

这个方案有点类似下一章我们会讨论到的水平数据划分，但它的优势在于主库包含了所有的数据集，这意味着无须为了一条写入查询去访问多个服务器。如果读操作无法在备库上找到数据，还可以通过主库来查询。即使不能从备库上读取所有数据，也可以移除大量的主库读负担。

最简单的方法是在主库上将数据划分到不同的数据库里。然后将每个数据库复制到不同的备库上。例如，若需要将公司的每一个部门的数据复制到不同的备库，可以创建名为 sales、marketing、procurement 等的数据库，每个备库通过选项 replicate_wild_do_table 选项来限制给定数据库的数据。下面是 sales 数据库的配置： 478

```
replicate_wild_do_table = sales.%
```

也可以通过一台分发主库进行分发。举个例子，如果想通过一个很慢或者非常昂贵的网络，从一台负载很高的数据库上复制一部分数据，就可以使用一个包含 blackhole 表和过滤规则的本地分发主库，分发主库可以通过复制过滤移除不需要的日志。这可以避免在主库上进行不安全的日志选项设定，并且无须传输所有的数据到远程备库。

分离功能

许多应用都混合了在线事务处理（OLTP）和在线数据分析（OLAP）的查询。OLTP 查询比较短并且是事务型的，OLAP 查询则通常很大，也很慢，并且不要求绝对最新的数据。这两种查询给服务器带来的负担完全不同，因此它们需要不同的配置，甚至可能使用不同的存储引擎或者硬件。

一个常见的办法是将 OLTP 服务器的数据复制到专门为 OLAP 工作负载准备的备库上。这些备库可以有不同的硬件、配置、索引或者不同的存储引擎。如果决定在备库上执行 OLAP 查询，就可能需要忍受更大的复制延迟或降低备库的服务质量。这意味着在一个非专用的备库上执行一些任务时，可能会导致不可接受的性能，例如执行一条长时间运行的查询。

无须做一些特殊的配置，除了需要选择忽略主库上的一些数据，前提是能获得明显的提升。即使通过复制过滤器过滤掉一小部分的数据也会减少 I/O 和缓存活动。

数据归档

可以在备库上实现数据归档，也就是说可以在备库上保留主库上删除过的数据，在主库上通过 delete 语句删除数据是确保 delete 语句不传递到备库就可以实现。有两种通常的办法：一种是在主库上选择性地禁止二进制日志，另一种是在备库上使用 replicate_ignore_db 规则（是的，两种方法都很危险）。

479 第一种方法需要先将 SQL_LOG_BIN 设置为 0，然后再进行数据清理。这种方法的好处是不需要在备库进行任何配置，由于 SQL 语句根本没有记录到二进制日志中，效率会稍微有所提升。最大缺点也正因为没有将在主库的修改记录下来，因此无法使用二进制日志来进行审计或者做按时间点的数据恢复。另外还需要 SUPER 权限。

第二种方法是在清理数据之前对主库上特定的数据库使用 USE 语句。例如，可以创建一个名为 purge 的数据库，然后在备库的 *my.cnf* 文件里设置 replicate_ignore_db=purge 并重启服务器。备库将会忽略使用了 USE 语句指定的数据库。这种方法没有第一种方法的缺点，但有另一个小小的缺点：备库需要去读取它不需要的事件。另外，也可能有人在 purge 数据库上执行非清理查询，从而导致备库无法重放该事件。

Percona Toolkit 中的 pt-archiver 支持以上两种方式。

> 第三种办法是利用 binlog_ignore_db 来过滤复制事件。但正如之前提到的，这是一种很危险的操作。

将备库用作全文检索

许多应用要求合并事务和全文检索。然而在写作本书时，仅有 MyISAM 支持全文检索，但是 MyISAM 不支持事务（在 MySQL 5.6 有一个实验室预览版本实现了 InnoDB 的全文检索，但尚未 GA）。一个普遍的做法是配置一台备库，将某些表设置为 MyISAM 存储引擎，然后创建全文索引并执行全文检索查询。这避免了在主库上同时使用事务型和非事务型存储引擎所带来的复制问题，减轻了主库维护全文索引的负担。

只读备库

许多机构选择将备库设置为只读，以防止在备库进行的无意识修改导致复制中断。可以通过设置 read_only 选项来实现。它会禁止大部分写操作，除了复制线程和拥有超级权

限的用户以及临时表操作。只要不给也不应该给普通用户超级权限，这应该是很完美的方法。

模拟多主库复制

480

当前 MySQL 不支持多主库复制（一个备库拥有多个主库）。但是可以通过把一台备库轮流指向多台主库的方式来模拟这种结构。例如，可以先将备库指向主库 A，运行片刻，再将其指向主库 B 并运行片刻，然后再次切换回主库 A。这种办法的效果取决于数据以及两台主库导致备库所需完成的工作量。如果主库的负载很低，并且主库之间不会产生更新冲突，就会工作得很好。

需要做一些额外的工作来为每个主库跟踪二进制日志坐标。可能还需要保证备库的 I/O 线程在每一次循环提取超过需要的数据，否则可能会因为每次循环反复地提取和抛弃大量数据导致主库的网络流量和开销明显增大。

还可以使用主－主（或者环形）复制结构以及使用 blackhole 存储引擎表的备库来进行模拟，如图 10-13 所示。

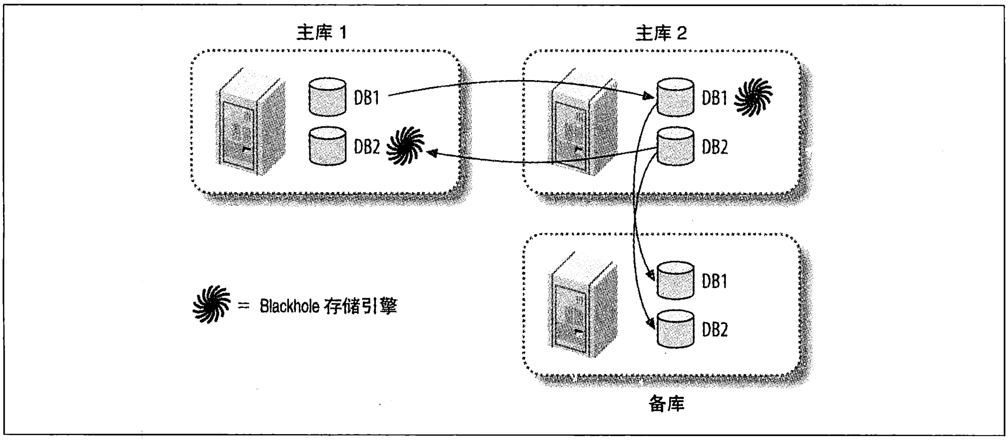


图10-13：使用双主结构和blackhole存储引擎表模拟多主复制

在这种配置中，两台主库拥有自己的数据，但也包含了对方的表，但是对方的表使用 blackhole 存储引擎以避免在其中存储实际数据。备库和其中任意一个主库相连都可以。备库不使用 blackhole 存储引擎，因此其对两个主库而言都是有效的。

事实上并不一定需要主－主拓扑结构来实现，可以简单地将 server1 复制到 server2，再从 server2 复制到备库。如果在 server2 上为从 server1 上复制的数据使用 blackhole 存储引擎，就不会包含任何 server1 的数据，如图 10-14 所示。

481

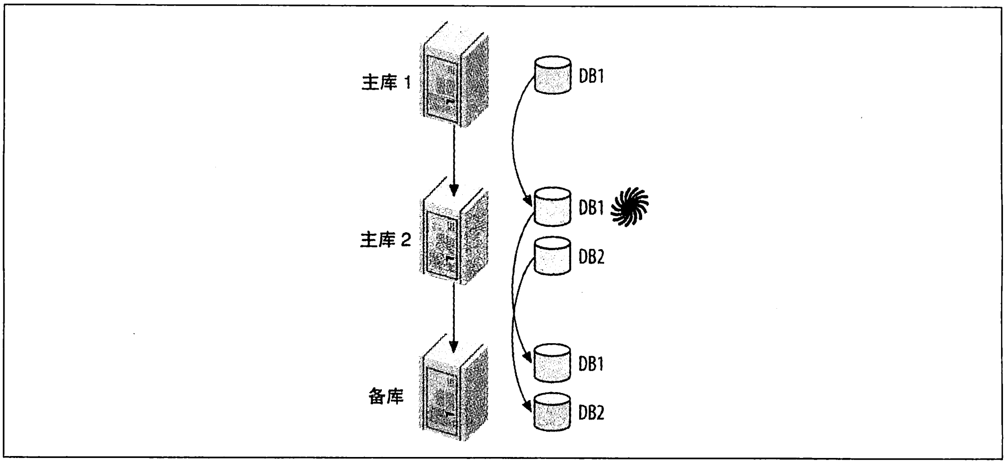


图10-14：另一种模拟多主复制的方法

这些配置方法常常会碰到一些常见的问题，例如，更新冲突或者建表时明确指定存储引擎。

另外一个选择是使用 Continuent 的 Tungsten Replicator，我们会在本章稍后部分讨论。

创建日志服务器

使用 MySQL 复制的另一种用途就是创建没有数据的日志服务器。它唯一的目的就是更加容易重放并且/或者过滤二进制日志事件。就如本章稍后所述，它对崩溃后重启复制很有帮助。同时对基于时间点的恢复也很有帮助，在第 15 章我们会讨论。

假设有一组二进制日志或中继日志——可能从备份或者一台崩溃的服务器上获取——希望能够重放这些日志中的事件，可以通过 *mysqlbinlog* 工具从其中提取出事件，但更加方便和高效的方法是配置一个没有任何数据的 MySQL 实例并使其认为这些二进制日志是它拥有的。如果只是临时需要，可以从 *http://mysqlsandbox.net* 上获得一个 MySQL 沙箱脚本来创建日志服务器。因为无须执行二进制日志，日志服务器也就不需要任何数据。它的目的仅仅是将数据提供给别的服务器（但复制账户还是需要的）。

我们来看看该策略是如何工作的（稍后会展示一些相关应用）。假设日志被命名为 *somelog-bin.000001*、*somelog-bin.000002*，等等，将这些日志放到日志服务器的日志文件夹中，假设为 */var/log/mysql*。然后在启动服务器前编辑 *my.cnf* 文件，如下所示：

```
log_bin       = /var/log/mysql/somelog-bin
log_bin_index = /var/log/mysql/somelog-bin.index
```

482 服务器不会自动发现日志文件，因此还需要更新日志的索引文件。下面这个命令可以在

类 UNIX 系统上完成[注14]。

```
# /bin/ls -1 /var/log/mysql/somelog-bin.[0-9]* > /var/log/mysql/somelog-bin.index
```

确保运行 MySQL 的账户能够读写日志索引文件。现在可以启动日志服务器并通过 SHOW MASTER LOGS 命令来确保其找到日志文件。

为什么使用日志服务器比用 *mysqlbinlog* 来实现恢复更好呢？有以下几个原因：

- 复制作为应用二进制日志的方法已经被大量的用户所测试，能够证明是可行的。*mysqlbinlog* 并不能确保像复制那样工作，并且可能无法正确生成二进制日志中的数据更新。
- 复制的速度更快，因为无须将语句从日志导出来并传送给 MySQL。
- 可以很容易观察到复制过程。
- 能够更方便处理错误。例如，可以跳过执行失败的语句。
- 更方便过滤复制事件。
- 有时候 *mysqlbinlog* 会因为日志记录格式更改而无法读取二进制日志。

10.5 复制和容量规划

写操作通常是复制的瓶颈，并且很难使用复制来扩展写操作。当计划为系统增加复制容量时，需要确保进行了正确的计算，否则很容易犯一些复制相关的错误。

例如，假设工作负载为 20% 的写以及 80% 的读。为了计算简单，假设有以下前提：

- 读和写查询包含同样的工作量。
- 所有的服务器是等同的，每秒能进行 1 000 次查询。
- 备库和主库有同样的性能特征。
- 可以把所有的读操作转移到备库。

如果当前有一个服务器能支持每秒 1 000 次查询，那么应该增加多少备库才能处理当前两倍的负载，并将所有的读查询分配给备库？

看上去应该增加两个备库并将 1 600 次读操作平分给它们。但是不要忘记，写入负载同样增加到了 400 次每秒，并且无法在主备服务器之间进行分摊。每个备库每秒必须处理 400 次写入，这意味着每个备库写入占了 40%，只能每秒为 600 次查询提供服务。因此，需要三台而不是两台备库来处理双倍负载。 483

如果负载再增加一倍呢？将有每秒 800 次写入，这时候主库还能处理，但备库的写入同

注 14： 我们明确地使用 /bin/ls 以避免启用通用别名，它们会为终端着色添加转义码。

样也提升到 80%，这样就需要 16 台备库来处理每秒 3 200 次读查询。并且如果再增加一点负载，主库也会无法承担。

这远远不是线性扩展，查询数量增加 4 倍，却需要增加 17 倍的服务器。这说明当为单台主库增加备库时，将很快达到投入远高于回报的地步。这仅仅是基于上面的假设，还忽略了一些事情，例如，单线程的基于语句的复制常常导致备库容量小于主库。真实的复制配置比我们的理论计算还要更差。

10.5.1 为什么复制无法扩展写操作

糟糕的服务容量比例的根本原因是不能像分发读操作那样把写操作等同地分发到更多服务器上。换句话说，复制只能扩展读操作，无法扩展写操作。

你可能想知道到底有没有办法使用复制来增加写入能力。答案是否定的，根本不行。对数据进行分区是唯一可以扩展写入的方法，我们在下一章会讲到。

一些读者可能会想到使用主 - 主拓扑结构（参阅前面介绍的“主动 - 主动模式下的主 - 主复制”）并为两个服务器执行写操作。这种配置比主备结构能支持稍微多一点的写入，因为可以在两台服务器之间共享串行化带来的开销。如果每台服务器上执行 50% 的写入，那复制的执行量也只有 50% 需要串行化。理论上讲，这比在一台机器上（主库）对 100% 的写入并发执行，而在另外一台机器（备库）上对 100% 的写入做串行化要更优。

这可能看起来很吸引人，然而这种配置还比不上单台服务器能支持的写入。一个有 50% 的写入被串行化的服务器性能比一台全部写入都并行化的服务器性能要低。

这是这种策略不能扩展写入的原因。它只能在两台服务器间共享串行化写入的缺点。所以“链中最弱的一环”并不是那么弱，它只提供了比主动 - 被动复制稍微好点的性能，但是增加了很大的风险，通常不能带来任何好处，具体原因见下一节。

484
10.5.2 备库什么时候开始延迟

一个关于备库比较普遍的问题是如何预测备库会在何时跟不上主库。很难去描述备库使用的复制容量为 5% 与 95% 的区别，但是至少能够在接近饱和前预警并估计复制容量。

首先应该观察复制延迟的尖刺。如果有复制延迟的曲线图，需要注意到图上的一些短暂的延迟骤升，这时候可能负载加大，备库短时间内无法跟上主库。当负载接近耗尽备库的容量时，会发现曲线上的凸起会更高更宽。前面曲线的上升角度不变，但随后当备库在产生延迟后开始追赶主库时，将会产生一个平缓的斜坡。这些突起的出现和增长是一个警告信息，意味着已经接近容量限制。

为了预测在将来的某个时间点会发生什么，可以人为地制造延迟，然后看多久备库能赶上主库。目的是为了明确地说明曲线上的斜坡的陡度。如果将备库停止一个小时，然后开启并在 1 小时内追赶上，说明正常情况下只消耗了一半的容量。也就是说，如果中午 12:00 停止备库复制，在 1:00 开启，并且在 2:00 追赶上，备库在一小时内完成了两个小时内所有的变更，说明复制可以在双倍速度下运行。

最后，如果使用的是 Percona Server 或者 MariaDB，也可以直接获取复制的利用率。打开服务器变量 userstat，然后执行如下语句：

```
mysql> SELECT * FROM INFORMATION_SCHEMA.USER_STATISTICS
    -> WHERE USER='#mysql_system#'\G
*************************** 1. row ***************************
                  USER: #mysql_system#
     TOTAL_CONNECTIONS: 1
CONCURRENT_CONNECTIONS: 2
        CONNECTED_TIME: 46188
             BUSY_TIME: 719
          ROWS_FETCHED: 0
          ROWS_UPDATED: 1882292
       SELECT_COMMANDS: 0
       UPDATE_COMMANDS: 580431
        OTHER_COMMANDS: 338857
   COMMIT_TRANSACTIONS: 1016571
 ROLLBACK_TRANSACTIONS: 0
```

可以将 BUSY_TIME 和 CONNECTED_TIME 的一半（因为备库有两个复制线程）做比较，来观察备库线程实际执行命令所花费的时间[注 15]。在我们的例子里，备库大约使用了其 3% 的能力，这并不意味着它不会遇到偶然的延迟尖刺——如果主库运行了一个超过 10 分钟才完成的变更，可能延迟的时间和变更执行的时间是相同的——但这很好地暗示了备库能够很快从一个延迟尖刺中恢复。 485

10.5.3 规划冗余容量

在构建一个大型应用时，有意让服务器不被充分使用，这应该是一种聪明并且划算的方式，尤其在使用复制的时候。有多余容量的服务器可以更好地处理负载尖峰，也有更多能力处理慢速查询和维护工作（如 OPTIMIZE TABLE 操作），并且能够更好地跟上复制。

试图同时向主 - 主拓扑结构的两个节点写入来减少复制问题通常是不划算的。分配给每台机器的读负载应该低于 50%，否则，如果某台服务器失效，就没有足够的容量了。如果两台服务器都能够独立处理负载，就用不着担心复制的问题了。

注 15： 如果复制线程总是在运行，你可以使用服务器的 uptime 来代替 CONNECTED_TIME 的一半。

构建冗余容量也是实现高可用性的最佳方式之一，当然还有别的方式，例如，当错误发生时让应用在降级模式下运行，第 12 章会介绍更多的细节。

10.6 复制管理和维护

配置复制一般来说不会是需要经常做的工作，除非有很多服务器。但是一旦配置了复制，监控和管理复制拓扑应该成为一项日常工作，不管有多少服务器。

这些工作应该尽量自动化，但不一定需要自己写工具来实现：在 16 章我们讨论了几个 MySQL 工具，其中许多都拥有内建的监控复制的能力或插件。

10.6.1 监控复制

复制增加了 MySQL 监控的复杂性。尽管复制发生在主库和备库上，但大多数工作是在备库上完成的，这也正是最常出问题的地方。是否所有的备库都在工作？最慢的备库延迟是多大？ MySQL 本身提供了大量可以回答上述问题的信息，但要实现自动化监控过程以及使复制更健壮还是需要用户做更多的工作。

486 在主库上，可以使用 SHOW MASTER STATUS 命令来查看当前主库的二进制日志位置和配置（详细参阅前面介绍的“配置主库和备库”部分）。还可以查看主库当前有哪些二进制日志是在磁盘上的：

```
mysql> SHOW MASTER LOGS;
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| mysql-bin.000220 |    425605 |
| mysql-bin.000221 |   1134128 |
| mysql-bin.000222 |     13653 |
| mysql-bin.000223 |     13634 |
+------------------+-----------+
```

该命令用于给 PURGE MASTER LOGS 命令决定使用哪个参数。另外还可以通过 SHOW BINLOG EVENTS 来查看复制事件。例如，在运行前一个命令后，我们在另一个不曾使用过的服务器上创建一个表，因为知道这是唯一改变数据的语句，而且也知道语句在二进制日志中的偏移量是 13634，所以我们可以看到如下内容：

```
mysql> SHOW BINLOG EVENTS IN 'mysql-bin.000223' FROM 13634\G
*************************** 1. row ***************************
   Log_name: mysql-bin.000223
        Pos: 13634
 Event_type: Query
  Server_id: 1
End_log_pos: 13723
       Info: use `test`; CREATE TABLE test.t(a int)
```

10.6.2 测量备库延迟

一个比较普遍的问题是如何监控备库落后主库的延迟有多大。虽然 SHOW SLAVE STATUS 输出的 Seconds_behind_master 列理论上显示了备库的延时，但由于各种各样的原因，并不总是准确的：

- 备库 Seconds_behind_master 值是通过将服务器当前的时间戳与二进制日志中的事件的时间戳相对比得到的，所以只有在执行事件时才能报告延迟。
- 如果备库复制线程没有运行，就会报延迟为 NULL。
- 一些错误（例如主备的 max_allowed_packet 不匹配，或者网络不稳定）可能中断复制并且/或者停止复制线程，但 Seconds_behind_master 将显示为 0 而不是显示错误。
- 即使备库线程正在运行，备库有时候可能无法计算延时。如果发生这种情况，备库会报 0 或者 NULL。
- 一个大事务可能会导致延迟波动，例如，有一个事务更新数据长达一个小时，最后提交。这条更新将比它实际发生时间要晚一个小时才记录到二进制日志中。当备库执行这条语句时，会临时地报告备库延迟为一个小时，然后又很快变成 0。 487
- 如果分发主库落后了，并且其本身也有已经追赶上它的备库，备库的延迟将显示为 0，而事实上和源主库之间是有延迟的。

解决这些问题的办法是忽略 Seconds_behind_master 的值，并使用一些可以直接观察和衡量的方式来监控备库延迟。最好的解决办法是使用 *heartbeat record*，这是一个在主库上会每秒更新一次的时间戳。为了计算延时，可以直接用备库当前的时间戳减去心跳记录的值。这个方法能够解决刚刚我们提到的所有问题，另外一个额外的好处是我们还可以通过时间戳知道备库当前的复制状况。包含在 Percona Toolkit 里的 *pt-heartbeat* 脚本是“复制心跳”最流行的一种实现。

心跳还有其他好处，记录在二进制日志中的心跳记录拥有许多用途，例如在一些很难解决的场景下可以用于灾难恢复。

我们刚刚所描述的几种延迟指标都不能表明备库需要多长时间才能赶上主库。这依赖于许多因素，例如备库的写入能力以及主库持续写入的次数。关于这个话题，详细参阅前面介绍的“何时备库开始延迟”。

10.6.3 确定主备是否一致

在理想情况下，备库和主库的数据应该是完全一样的。但事实上备库可能发生错误并导致数据不一致。即使没有明显的错误，备库同样可能因为 MySQL 自身的特性导致数据

不一致，例如 MySQL 的 Bug、网络中断、服务器崩溃，非正常关闭或者其他一些错误。[注16]

按照我们的经验来看，主备一致应该是一种规范，而不是例外，也就是说，检查你的主备一致性应该是一个日常工作，特别是当使用备库来做备份时尤为重要，因为你肯定不希望从一个已经损坏的备库里获得备份数据。

MySQL 并没有内建的方法来比较一台服务器与别的服务器的数据是否相同。它提供了
488 一些组件来为表和数据生成校验值，例如 CHECKSUM TABLE。但当复制正在进行时，这种方法是不可行的。

Percona Toolkit 里的 *pt-table-checksum* 能够解决上述几个问题。其主要特性是用于确认备库与主库的数据是否一致。工作方式是通过在主库上执行 INSERT...SELECT 查询。

这些查询对数据进行校验并将结果插入到一个表中。这些语句通过复制传递到备库，并在备库执行一遍，然后可以比较主备上的结果是否一样。由于该方法是通过复制工作的，它能够给出一致的结果而无须同时把主备上的表都锁上。

通常情况下可以在主库上运行该工具，参数如下：

```
$ pt-table-checksum --replicate=test.checksum <master_host>
```

该命令将检查所有的表，并将结果插入到 *test.checksum* 表中。当查询在备库执行完后，就可以简单地比较主备之间的不同了。*pt-table-checksum* 能够发现服务器所有的备库，在每台备库上运行查询，并自动地输出结果。在写作本书时，*pt-table-checksum* 是唯一能够有效地比较主备一致性的工具。

10.6.4 从主库重新同步备库

在你的职业生涯中，也许会不止一次需要去处理未被同步的备库。可能是使用校验工具发现了数据不一致，或是因为已经知道是备库忽略了某条查询或者有人在备库上修改了数据。

传统的修复不一致的办法是关闭备库，然后重新从主库复制一份数据。当备库数据不一致的问题可能导致严重后果时，一旦发现就应该将备库停止并从生产环境移除，然后再从一个备份中克隆或恢复备库。

这种方法的缺点是不太方便，特别是数据量很大时。如果能够找出并修复不一致的数据，要比从其他服务器上重新克隆数据要有效得多。如果发现的不一致并不严重，就可以保持备库在线，并重新同步受影响的数据。

注 16： 如果你正在使用非事务型存储引擎，不首先调用 STOP SLAVE 就关闭服务器是很不妥当的。

最简单的办法是使用 *mysqldump* 转储受影响的数据并重新导入。在整个过程中，如果数据没有发生变化，这种方法会很好。你可以在主库上简单地锁住表然后进行转储，再等待备库赶上主库，然后将数据导入到备库中。（需要等待备库赶上主库，这样就不至于为其他表引入新的不一致，例如那些可能通过和失去同步的表做 join 后进行数据更新的表）。

虽然这种方法在许多场景下是可行的，但在一个繁忙的服务器上有可能行不通。另外一个缺点是在备库上通过非复制的方式改变数据。通过复制改变备库数据（通过在主库上执行更新）通常是一种安全的技术，因为它避免了竞争条件和其他意料外的事情。如果表很大或者网络带宽受限，转储和重载数据的代价依然很高。当在一个有一百万行的表上只有一千行不同的数据呢？转储和重载表的数据是非常浪费资源的。 489

pt-table-sync 是 Percona Toolkit 中的另外一个工具，可以解决该问题。该工具能够高效地查找并解决表之间的不同。它同样通过复制工作，在主库上执行查询，在备库上重新同步，这样就没有竞争条件。它是结合 *pt-table-checksum* 生成的 checksum 表来工作的，所以只能操作那些已知不同步的表的数据块。但该工具不是在所有场景下都有效。为了正确地同步主库和备库，该工具要求复制是正常的，否则就无法工作。*pt-table-sync* 设计得很高效，但当数据量非常大时效率还是会很低。比较主库和备库上 1TB 的数据不可避免地会带来额外的工作。尽管如此，在那些合适的场景中，该工具依然能节约大量的时间和工作。

10.6.5 改变主库

迟早会有把备库指向一个新的主库的需求。也许是为了更迭升级服务器，或者是主库出现问题时需要把一台备库转换成主库，或者只是希望重新分配容量。不管出于什么原因，都需要告诉其他的备库新主库的信息。

如果这是计划内的操作，会比较容易（至少比紧急情况下要容易）。只需在备库简单地使用 `CHANGE MASTER TO` 命令，并指定合适的值。大多数值都是可选的。只需要指定需要改变的项即可。备库将抛弃之前的配置和中继日志并从新的主库开始复制。同样新的参数会被更新到 *master.info* 文件中，这样就算重启，备库配置信息也不会丢失。

整个过程中最难的是获取新主库上合适的二进制日志位置，这样备库才可以从和老主库相同的逻辑位置开始复制。

把备库提升为主库要更困难一点。有两种场景需要将备库替换为主库，一种是计划内的提升，一种是计划外的提升。

490

计划内的提升

把备库提升为主库理论上是很简单的。简单来说，有以下步骤：

1. 停止向老的主库写入。
2. 让备库追赶上主库（可选的，会简化下面的步骤）。
3. 将一台备库配置为新的主库。
4. 将备库和写操作指向新的主库，然后开启主库的写入。

但这其中还隐藏着很多细节。一些场景可能依赖于复制的拓扑结构。例如，主－主结构和主－备结构的配置就有所不同。

更深入一点，下面是大多数配置需要的步骤：

1. 停止当前主库上的所有写操作。如果可以，最好能将所有的客户端程序关闭（除了复制连接）。为客户端程序建立一个“do not run”这样的类似标记可能会有所帮助。如果正在使用虚拟 IP 地址，也可以简单地关闭虚拟 IP，然后断开所有的客户端连接以关闭其打开的事务。
2. 通过 `FLUSH TABLES WITH READ LOCK` 在主库上停止所有活跃的写入，这一步是可选的。也可以在主库上设置 `read_only` 选项。从这一刻开始，应该禁止向即将被替换的主库做任何写入。因为一旦它不是主库，写入就意味着数据丢失。注意，即使设置 `read_only` 也不会阻止当前已存在的事务继续提交。为了更好地保证这一点，可以“kill”所有打开的事务，这将会真正地结束所有写入。
3. 选择一个备库作为新的主库，并确保它已经完全跟上主库（例如，让它执行完所有从主库获得的中继日志）。
4. 确保新主库和旧主库的数据是一致的。可选。
5. 在新主库上执行 `STOP SLAVE`。
6. 在新主库上执行 `CHANGE MASTER TO MASTER_HOST=''`，然后再执行 `RESET SLAVE`，使其断开与老主库的连接，并丢弃 *master.info* 里记录的信息（如果连接信息记录在 *my.cnf* 里，会无法正确工作，这也是我们建议不要把复制连接信息写到配置文件里的原因之一）。
7. 执行 `SHOW MASTER STATUS` 记录新主库的二进制日志坐标。
8. 确保其他备库已经追赶上。
9. 关闭旧主库。
10. 在 MySQL 5.1 及以上版本中，如果需要，激活新主库上事件。
11. 将客户端连接到新主库。

12. 在每台备库上执行 CHANGE MASTER TO 语句，使用之前通过 SHOW MASTER STATUS 获得的二进制日志坐标，来指向新的主库。 491

> 当将备库提升为主库时，要确保备库上任何特有的数据库、表和权限已经被移除。可能还需要修改备库特有的配置选项，例如 innodb_flush_log_at_trx_commit 选项。同样的，如果是把主库降级为备库，也要保证进行需要的配置。
>
> 如果主备的配置相同，就不需要做任何改变。

计划外的提升

当主库崩溃时，需要提升一台备库来代替它，这个过程可能就不太容易。如果只有一台备库，可以直接使用这台备库。但如果有超过一台的备库，就需要做一些额外的工作。

另外，还有潜在的丢失复制事件的问题。可能有主库上已发生的修改还没有更新到它的任何一台备库上的情况。甚至还可能一条语句在主库上执行了回滚，但在备库上没有回滚，这样备库可能超过主库的逻辑复制位置[注17]。如果能在某一点恢复主库的数据，也许就可以取得丢失的语句并手动执行它们。

在以下步骤中，需要确保在计算中使用 Master_Log_File 和 Read_Master_Log_Pos 的值。以下是对主备拓扑结构中的备库进行提升的过程：

1. 确定哪台备库的数据最新。检查每台备库上 SHOW SLAVE STATUS 命令的输出，选择其中 Master_Log_File/read_Master_Log_Pos 的值最新的那个。
2. 让所有备库执行完所有其从崩溃前的旧主库那获得的中继日志。如果在未完成前修改备库的主库，它会抛弃剩下的日志事件，从而无法获知该备库在什么地方停止。
3. 执行前一小节的 5 ~ 7 步。
4. 比较每台备库和新主库上的 Master_Log_File/Read_Master_Log_Pos 的值。
5. 执行前一小节的 10 ~ 12 步。

正如本章开始我们推荐的，假设已经在所有的备库上开启了 log_bin 和 log_slave_ 492
updates，这样可以帮助你将所有的备库恢复到一个一致的时间点，如果没有开启这两个选项，则不能可靠地做到这一点。

注 17： 这是有可能的，即使 MySQL 在事务提交前并不记录任何事件。具体参阅“混合事务型和非事务型表”。另外一种场景是主库崩溃后恢复，但没有设置 innodb_flush_log_at_trx_commit 的值为 1，所以可能丢失一些更新。

确定期望的日志位置

如果有备库和新主库的位置不相同，则需要找到该备库最后一条执行的事件在新主库的二进制日志中相应的位置，然后再执行 CHANGE MASTER TO。可以通过 *mysqlbinlog* 工具来找到备库执行的最后一条查询，然后在主库上找到同样的查询，进行简单的计算即可得到。

为了便于描述，假设每个日志事件有一个自增的数字 ID，最新的备库，也就是新主库，在旧主库崩溃时获得了编号为 100 的事件，假设有另外两台备库：replica2 和 replica3。replica2 已经获取了 99 号事件，replica3 获取了 98 号事件。如果把两台备库都指向新主库的同一个二进制日志位置，它们将从 101 号事件开始复制，从而导致数据不同步。但只要新主库的二进制日志已经通过 log_slave_updates 打开，就可以在新主库的二进制日志中找到 99 号和 100 号日志，从而将备库恢复到一致的状态。

由于服务器重启，不同的配置，日志轮转或者 FLUSH LOGS 命令，同一个事件在不同的服务器上可能有不同的偏移量。找到这些事件可能会耗时很长并且枯燥，但是通常没有难度。通过 *mysqlbinlog* 从二进制日志或中继日志中解析出每台备库上执行的最后一个事件，并同样使用该命令解析新主库上的二进制日志，找到相同的查询，*mysqlbinlog* 会打印出该事件的偏移量，在 CHANGE MASTER TO 命令中使用这个值[注18]。

更快的方法是把新主库和停止的备库上的字节偏移量相减，它显示了字节位置的差异。然后把这个值和新主库当前二进制日志的位置相减，就可以得到期望的查询的位置。只需要验证一下就可以据此启动备库。

让我们看看一个相关的例子，假设 server1 是 server2 和 server3 的主库，其中服务器 server1 已经崩溃。根据 SHOW SLAVE STATUS 获得 Master_Log_File/Read_Master_Log_Pos 的值，server2 已经执行完了 server1 上所有的二进制日志，但 server3 还不是最新数据。图 10-15 显示了这个场景（日志事件和偏移量仅仅是为了举例）。

正如图 10-15 所示，我们可以肯定 server2 已经执行完了主库上的所有二进制日志，因
493
为 Master_Log_File 和 Read_Master_Log_Pos 值和 server1 上最后的日志位置是相吻合的，因此我们可以将 server2 提升为新主库，并将 server3 设置为 server2 的备库。

应该在 server3 上为需要执行的 CHANGE MASTER TO 语句赋予什么样的参数呢？这里需要做一点点计算和调查。server3 在偏移量 1493 停止，比 server2 执行的最后一条语句的偏移量 1582 要小 89 字节。server2 正在向偏移量为 8167 的二进制日志写入，8167−89=8078，因此理论上我们应该将 server3 指向 server2 的日志的偏移量为 8078 的位置。最

注 18： 正如前面提到的，*pt-heartbeat* 的心跳记录能够很好地帮助你找到你正在查找的事件的大约位置。

好去确认下这个位置附近的日志事件，以确定在该位置上是否是正确的日志事件，因为可能有别的例外，例如有些更新可能只发生在 server2 上。

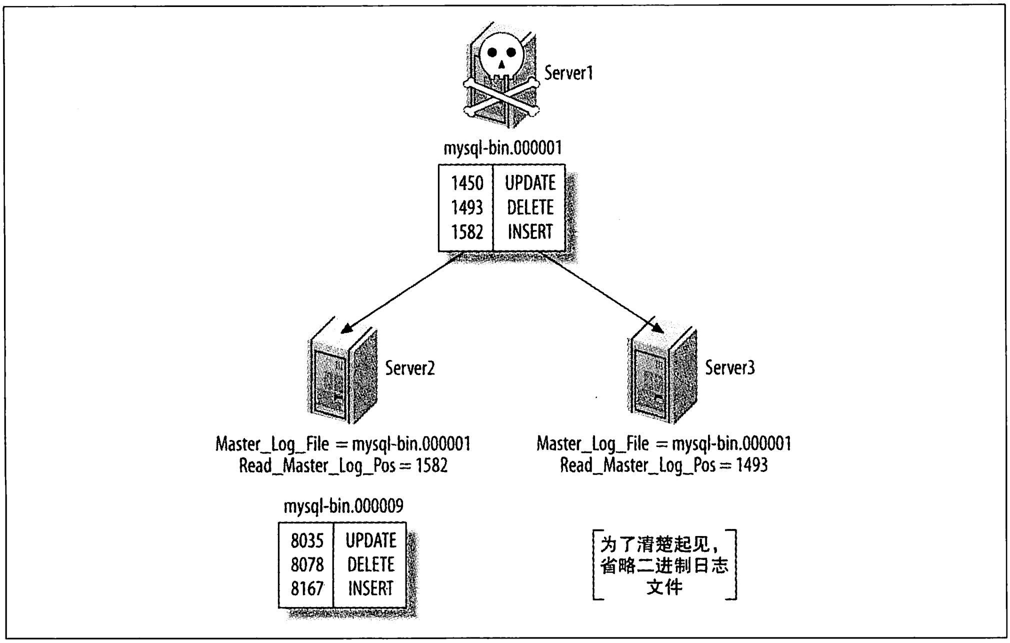


图10-15：当server1崩溃，server2已追赶上，但server3的复制落后

假设我们观察到的事件是一样的，下面这条命令会将 server3 切换为 server2 的备库。

```
server2> CHANGE MASTER TO MASTER_HOST="server2", MASTER_LOG_FILE="mysql-bin.000009",
         MASTER_LOG_POS=8078;
```

如果服务器在它崩溃时已经执行完成并记录了超过一个事件，会怎么样呢？因为 server2 仅仅读取并执行到了偏移位置 1582，你可能永远地失去了一个事件。但是如果老主库的磁盘没有损坏，仍然可以通过 *mysqlbinlog* 或者从日志服务器的二进制日志中找到丢失的事件。

如果需要从老主库上恢复丢失的事件，建议在提升新主库之后且在允许客户端连接之前做这件事情。这样就无须在每台备库上都执行丢失的事件，只需使用复制来完成。但如果崩溃的老主库完全不可用，就不得不等待，稍后再做这项工作。 494

上述流程中一个可调整的地方是使用可靠的方式来存储二进制日志，如 SAN 或分布式复制数据库设备（DRBD）。即使主库完全失效，依然能够获得它的二进制日志。也可以

设置一个日志服务器，把备库指向它，然后让所有备库赶上主库失效的点。这使得提升一个备库为新的主库没那么重要，本质上这和计划中的提升是相同的。我们将在下一章进一步讨论这些存储选项。

> 当提升一台备库为主库时，千万不要将它的服务器 ID 修改成原主库的服务器 ID，否则将不能使用日志服务器从一个旧主库来重放日志事件。这也是确保服务器 ID 最好保持不变的原因之一。

10.6.6 在一个主 - 主配置中交换角色

主 - 主复制拓扑结构的一个好处就是可以很容易地切换主动和被动的角色，因为其配置是对称的。本小节介绍如何完成这种切换。

当在主 - 主配置下切换角色时，必须确保任何时候只有一个服务器可以写入。如果两台服务器交叉写入，可能会导致写入冲突。换句话说，在切换角色后，原被动服务器不应该接收到主动服务器的任何二进制日志。可以通过确保原被动服务器的复制 SQL 线程在该服务器可写之前已经赶上主动服务器来避免。

通过以下步骤切换服务器角色，可以避免更新冲突的危险：

1. 停止主动服务器上的所有写入。
2. 在主动服务器上执行 `SET GLOBAL read_only=1`，同时在配置文件里也设置一下 `read_only`，防止重启后失效。但记住这不会阻止拥有超级权限的用户更改数据。如果想阻止所有人更改数据，可以执行 `FLUSH TABLES WITH READ LOCK`。如果没有这么做，你必须 kill 所有的客户端连接以保证没有长时间运行的语句或者未提交的事务。
3. 在主动服务器上执行 `SHOW MASTER STATUS` 并记录二进制日志坐标。
4. 495 使用主动服务器上的二进制日志坐标在被动服务器上执行 `SELECT MASTER_POS_WAIT()`。该语句将阻塞住，直到复制跟上主动服务器。
5. 在被动服务器上执行 `SET GLOBAL read_only=0`，这样就变换成主动服务器。
6. 修改应用的配置，使其写入到新的主动服务器中。

可能还需要做一些额外的工作，包括更改两台服务器的 IP 地址，这取决于应用的配置，我们将在下一节讨论这个话题。

10.7 复制的问题和解决方案

中断 MySQL 的复制并不是件难事。因为实现简单，配置相当容易，但也意味着有很多方式会导致复制停止，陷入混乱并中断。本章描述了一些比较普遍的问题，讨论如何重现这些问题，以及当遇到这些问题时如何解决或者阻止其发生。

10.7.1 数据损坏或丢失的错误

由于各种各样的原因，MySQL 的复制并不能很好地从服务器崩溃、掉电、磁盘损坏、内存或网络错误中恢复。遇到这些问题时几乎可以肯定都需要从某个点开始重启复制。

大部分由于非正常关机后导致的复制问题都是由于没有把数据及时地刷到磁盘。下面是意外关闭服务器时可能会碰到的情况。

主库意外关闭

如果没有设置主库的 `sync_binlog` 选项，就可能在崩溃前没有将最后的几个二进制日志事件刷新到磁盘中。备库 I/O 线程因此也可一直处于读不到尚未写入磁盘的事件的状态中。当主库重新启动时，备库将重连到主库并再次尝试去读该事件，但主库会告诉备库没有这个二进制日志偏移量。二进制日志转储线程通常很快，因此这种情况并不经常发生。

解决这个问题的方法是指定备库从下一个二进制日志的开头读日志。但是一些日志事件将永久地丢失，建议使用 Percona Toolkit 中的 *pt-table-checksum* 工具来检查主备一致性，以便于修复。可以通过在主库开启 `sync_binlog` 来避免事件丢失。

即使开启了 `sync_binlog`，MyISAM 表的数据仍然可能在崩溃的时候损坏，对于 InnoDB 事务，如果 `innodb_flush_log_at_trx_commit` 没有设为 1，也可能丢失数据(但数据不会损坏)。

备库意外关闭 496

当备库在一次非计划中的关闭后重启时，会去读 *master.info* 文件以找到上次停止复制的位置。不幸的是，该文件并没有同步写到磁盘，文件中存储的信息可能是错误的。备库可能会尝试重新执行一些二进制日志事件，这可能会导致唯一索引错误。除非能确定备库在哪里停止（通常不太可能），否则唯一的办法就是忽略那些错误。Percona Toolkit 中的 *pt-slave-restart* 工具可以帮助完成这一点。

如果使用的都是 InnoDB 表，可以在重启后观察 MySQL 错误日志。InnoDB 在恢复过程中会打印出它的恢复点的二进制日志坐标。可以使用这个值来决定备库指向主库的偏移量。Percona Server 提供了一个新的特性，可以在恢复的过程中自动将这些信息提取出来，并更新 *master.info* 文件，从根本上使得复制能够协调好备库上的事务。

MySQL 5.5 也提供了一些选项来控制如何将 *master.info* 和其他文件刷新到磁盘，这有助于减少这些问题。

除了由于 MySQL 非正常关闭导致的数据丢失外，磁盘上的二进制日志或中继日志文件损坏并不罕见。下面是一些更普遍的场景：

主库上的二进制日志损坏

如果主库上的二进制日志损坏，除了忽略损坏的位置外你别无选择。可以在主库上执行 `FLUSH LOGS` 命令，这样主库会开始一个新的日志文件，然后将备库指向该文件的开始位置。也可以试着去发现损坏区域的结束位置。某些情况下可以通过 `SET GLOBAL SQL_SLAVE_SKIP_COUNTER = 1` 来忽略一个损坏的事件。如果有多个损坏的事件，就需要重复该步骤，直到跳过所有损坏的事件。但如果有太多的损坏事件，这么做可能就没有意义了。损坏的事件头会阻止服务器找到下一个事件。这种情况下，可能不得不手动地去找到下一个完好的事件。

备库上的中继日志损坏

如果主库上的日志是完好的，就可以通过 `CHANGE MASTER TO` 命令丢弃并重新获取损坏的事件。只需要将备库指向它当前正在复制的位置（`Relay_Master_Log_File`/`Exec_Master_Log_Pos`）。这会导致备库丢弃所有在磁盘上的中继日志。就这一点而言，MySQL 5.5 做了一些改进，它能够在崩溃后自动重新获取中继日志。

二进制日志与 InnoDB 事务日志不同步

当主库崩溃时，InnoDB 可能将一个事务标记为已提交，此时该事务可能还没有记录到二进制日志中。除非是某个备库的中继日志已经保存，否则没有任何办法恢复丢失的事务。在 MySQL 5.0 版本可以设置 `sync_binlog` 选项来防止该问题，对于更早的 MySQL 4.1 可以设置 `sync_binlog` 和 `safe_binlog` 选项。

497 当一个二进制日志损坏时，能恢复多少数据取决于损坏的类型，有几种比较常见的类型：

数据改变，但事件仍是有效的 SQL

不幸的是，MySQL 甚至无法察觉这种损坏。因此最好还是经常检查备库的数据是否正确。在 MySQL 未来的版本中可能会被修复。

数据改变并且事件是无效的 SQL

这种情况可以通过 *mysqlbinlog* 提取出事件并看到一些错乱的数据，例如：

```
UPDATE tbl SET col?????????????????
```

可以通过增加偏移量的方式来尝试找到下一个事件，这样就可以只忽略这个损坏的事件。

数据遗漏并且/或者事件的长度是错误的

这种情况下，*mysqlbinlog* 可能会发生错误退出或者直接崩溃，因为它无法读取事件，并且找不到下一个事件的开始位置。

某些事件已经损坏或被覆盖，或者偏移量已经改变并且下一个事件的起始偏移量也是错误的

同样的，这种情况下 *mysqlbinlog* 也起不了多少作用。

当损坏非常严重，通过 *mysqlbinlog* 已经无法获取日志事件时，就不得不进行一些十六进制的编辑或者通过一些烦琐的技术来找到日志事件的边界。这通常并不困难，因为有一些可辨识的标记会分割事件。

如下例所示，首先使用 *mysqlbinlog* 找到样例日志的日志事件偏移量：

```
$ mysqlbinlog mysql-bin.000113 | egrep '^# at '
# at 4
# at 98
# at 185
# at 277
# at 369
# at 447
```

一个找到日志偏移量的比较简单的方法是比较一下 *string* 命令输出的偏移量：

```
$ strings -n 2 -t d mysql-bin.000113
      1 binpC'G
     25 5.0.38-Ubuntu_0ubuntu1.1-log
     99 C'G
    146 std
    156 test
    161 create table test(a int)
    186 C'G
    233 std
    243 test
    248 insert into test(a) values(1)
    278 C'G
    325 std
    335 test
    340 insert into test(a) values(2)
    370 C'G
    417 std
    427 test
    432 drop table test
    448 D'G
    474 mysql-bin.000114
```
498

有一些可辨别的模式可以帮助定位事件的开头，注意以 'G 结尾的字符串在日志事件开头的一个字节后的位置。它们是固定长度的事件头的一部分。

这些值因服务器而异，因此结果也可能取决于解析的日志所在的服务器。简单地分析后

应该能够从二进制日志中找到这些模式并找到下一个完整的日志事件偏移量。然后通过 *mysqlbinlog* 的 *--start-position* 选项来跳过损坏的事件，或者使用 CHANGE MASTER TO 命令的 MASTER_LOG_POS 参数。

10.7.2 使用非事务型表

如果一切正常，基于语句的复制通常能够很好地处理非事务型表。但是当对非事务型表的更新发生错误时，例如查询在完成前被 kill，就可能导致主库和备库的数据不一致。

例如，假设更新一个 MyISAM 表的 100 行数据，若查询更新到了其中 50 条时有人 kill 该查询，会发生什么呢？一半的数据改变了，而另一半则没有，结果是复制必然不同步，因为该查询会在备库重放并更新完 100 行数据（MySQL 随后会在主库上发现查询引起的错误，而备库上则没有报错，此后复制将会发生错误并中断）。

如果使用的是 MyISAM 表，在关闭 MySQL 之前需要确保已经运行了 STOP SLAVE，否则服务器在关闭时会 kill 所有正在运行的查询（包括没有完成的更新）。事务型存储引擎则没有这个问题。如果使用的是事务型表，失败的更新会在主库上回滚并且不会记录到二进制日志中。

10.7.3 混合事务型和非事务型表

如果使用的是事务型存储引擎，只有在事务提交后才会将查询记录到二进制日志中。因此如果事务回滚，MySQL 就不会记录这条查询，也就不会在备库上重放。

499 但是如果混合使用事务型和非事务型表，并且发生了一次回滚，MySQL 能够回滚事务型表的更新，但非事务型表则被永久地更新了。只要不发生类似查询中途被 kill 这样的错误，这就不是问题：MySQL 此时会记录该查询并记录一条 ROLLBACK 语句到日志中。结果是同样的语句也在备库执行，所有的都很正常。这样效率会低一点，因为备库需要做一些工作并且最后再把它们丢弃掉。但理论上能够保证主备的数据一致。

目前看来一切很正常。但是如果备库发生死锁而主库没有也可能会导致问题。事务型表的更新会被回滚，而非事务型表则无法回滚，此时备库和主库的数据是不一致的。

防止该问题的唯一办法是避免混合使用事务型和非事务型表。如果遇到这个问题，唯一的解决办法是忽略错误，并重新同步相关的表。

基于行的复制不会受这个问题的影响。因为它记录的是数据的更改，而不是 SQL 语句。如果一条语句改变了一个 MyISAM 表和一个 InnoDB 表的某些行，然后主库上发生了一次死锁，InnoDB 表的更新会被回滚，而 MyISAM 表的更新仍会被记录到日志中并在备库重放。

10.7.4 不确定语句

当使用基于语句的复制模式时，如果通过不确定的方式更改数据可能会导致主备不一致。例如，一条带 LIMIT 的 UPDATE 语句更改的数据取决于查找行的顺序，除非能保证主库和备库上的顺序相同。例如，若行根据主键排序，一条查询可能在主库和备库上更新不同的行，这些问题非常微妙并且很难注意到。所以一些人禁止对那些会更新数据的语句使用 LIMIT。另外一种不确定的行为是在一个拥有多个唯一索引的表上使用 REPLACE 或者 INSERT IGNORE 语句——MySQL 在主库和备库上可能会选择不同的索引。

另外还要注意那些涉及 INFORMATION_SCHEMA 表的语句。它们很容易在主库和备库上产生不一致，其结果也会不同。最后，需要注意许多系统变量，例如 @@server_id 和 @@hostname，在 MySQL 5.1 之前无法正确地复制。

基于行的复制则没有上述限制。

10.7.5 主库和备库使用不同的存储引擎

500

正如本章之前提到的，在备库上使用不同的存储引擎，有时候可以带来好处。但是在一些场景下，当使用基于语句的复制方式时，如果备库使用了不同的存储引擎，则可能造成一条查询在主库和备库上的执行结果不同，例如不确定语句（如前一小节提到的）在主备库使用不同的存储引擎时更容易导致问题。

如果发现主库和备库的某些表已经不同步，除了检查更新这些表的查询外，还需要检查两台服务器上使用的存储引擎是否相同。

10.7.6 备库发生数据改变

基于语句的复制方式前提是确保备库上有和主库相同的数据，因此不应该允许对备库数据的任何更改（比较好的办法是设置 read_only 选项）。假设有如下语句：

```
mysql> INSERT INTO table1 SELECT * FROM table2;
```

如果备库上 table2 的数据和主库上不同，该语句会导致 table1 的数据也会不一致。换句话说，数据不一致可能会在表之间传播。不仅仅是 INSERT......SELECT 查询，所有类型的查询都可能发生。有两种可能的结果：备库上发生重复索引键冲突错误或者根本不提示任何错误。如果能报告错误还好，起码能够提示你主备数据已经不一致。无法察觉的不一致可能会悄无声息地导致各种严重的问题。

唯一的解决办法就是重新从主库同步数据。

10.7.7 不唯一的服务器 ID

这种问题更加难以捉摸。如果不小心为两台备库设置了相同的服务器 ID，看起来似乎没有什么问题，但如果查看错误日志，或者使用 *innotop* 查看主库，可能会看到一些古怪的信息。

在主库上，会发现两台备库中只有一台连接到主库（通常情况下所有的备库都会建立连接以等待随时进行复制）。在备库的错误日志中，则会发现反复的重连和连接断开信息，但不会提及被错误配置的服务器 ID。

MySQL 可能会缓慢地进行正确的复制，也可能无法进行正确复制，这取决于 MySQL 的版本，给定的备库可能会丢失二进制日志事件，或者重复执行事件，导致重复键错误（或
501 者不可见的数据损坏）。也可能因为备库的互相竞争造成主库的负载升高。如果备库竞争非常激烈，会导致错误日志在很短的时间内急剧增大。

唯一的解决办法是小心设置备库的服务器 ID。一个比较好的办法是创建一个主库到备库的服务器 ID 映射表，这样就可以跟踪到备库的 ID 信息[注19]。如果备库全在一个子网络内，可以将每台机器 IP 的后八位作为唯一 ID。

10.7.8 未定义的服务器 ID

如果没有在 *my.cnf* 里定义服务器 ID，可以通过 CHANGE MASTER TO 来设置备库，但却无法启动复制：

```
mysql> START SLAVE;
ERROR 1200 (HY000): The server is not configured as slave; fix in config file or with
CHANGE MASTER TO
```

这个报错可能会让人困惑，因为刚刚执行 CHANGE MASTER TO 设置了备库，并且通过 SHOW MASTER STATUS 也确认了。执行 SELECT @@server_id 也可以获得一个值，但这只是默认值，必须为备库显式地设置服务器 ID。

10.7.9 对未复制数据的依赖性

如果在主库上有备库不存在的数据库或表，复制会很容易意外中断，反之亦然。假设主库上有一个备库不存在的数据库，命名为 scratch。如果在主库上发生对该数据库中表的更新，备库会在尝试重放这些更新时中断。同样的，如果在主库上创建一个备库上已存在的表，复制也可能中断。

没有什么好的解决办法，唯一的办法就是避免在主库上创建备库上没有的表。

注 19： 也许你想把它保存在服务器中，这不完全是玩笑，可以给 ID 列添加一个唯一索引。

这样的表是如何创建的呢？有很多可能的方式，其中一些可能更难防范。例如，假设先在备库上创建一个数据库 scratch，该数据库在主库上不存在，然后因为某些原因切换了主备。当完成这些后，可能忘记了移除 scratch 数据库以及它的权限。这时候一些人就可以连接到该数据库并执行一些查询，或者一些定期的任务会发现这些表，并在每个表上执行 OPTIMIZE TABLE 命令。

当提升备库为主库时，或者决定如何配置备库时，需要注意这一点。任何导致主备不同的行为都会产生潜在的问题。

10.7.10 丢失的临时表

502

临时表在某些时候比较有用，但不幸的是，它与基于语句的复制方式是不相容的。如果备库崩溃或者正常关闭，任何复制线程拥有的临时表都会丢失。重启备库后，所有依赖于该临时表的语句都会失败。

当基于语句进行复制时，在主库上并没有安全使用临时表的方法。许多人确实很喜欢临时表，所以很难去说服他们，但这是不可否认的[注20]。不管它们的存在多么短暂，都会使得备库的启动和停止以及崩溃恢复变得困难，即使是在一个事务内使用也一样。（如果在备库使用临时表可能问题会少些，但如果备库本身也是一个主库，问题依然存在。）

如果备库重启后复制因找不到临时表而停止，可能需要做以下一些事情：可以直接跳过错误，或者手动地创建一个名字和结构相同的表来代替消失的临时表。不管用什么办法，如果写入查询依赖于临时表，都可能造成数据不一致。

避免使用临时表没有看起来那么难，临时表主要有两个比较有用的特性：

- 只对创建临时表的连接可见。所以不会和其他拥有相同名字临时表的连接起冲突。
- 随着连接关闭而消失，所以无须显式地移除它们。

可以保留一个专用的数据库，在其中创建持久表，把它们作为伪临时表，以模拟这些特性。只需要为它们选择一个唯一的名字。还好这很容易做到：简单地将连接 ID 拼接到表名之后。例如，之前创建临时表的语句为：CREATE TEMPORARY TABLE top_users(...)，现在则可以执行 CREATE TABLE temp.top_users_1234(...)，其中 1234 是函数 CONNECTION_ID() 的返回值。当应用不再使用该伪临时表后，可以将其删除或使用一个清理线程来将其移除。表名中使用连接 ID 可以用于确定哪些表不再被使用——可以通过 SHOW PROCESSLIST 命令来获得活跃连接列表，并将其与表名中的连接 ID 相比较[注21]。

注 20：我们已经有人尝试各种方法来解决这个问题，但对于基于语句的复制并没有安全的临时表创建方法。起码一段时期是这样，不管你如何认为，起码我们已经证明了这是不可行的。

注 21：*pt-find*——另一个 Percona Toolkit 工具——通过 *--connection-id* 和 *--server-id* 选项能够轻易地移除伪临时表。

503 使用实体表而非临时表还有别的好处。例如，能够帮助你更容易调试应用程序，因为可以通过别的连接来查看应用正在维护的数据。如果使用的是临时表，可能就没这么容易做到。

但是实体表可能会比临时表多一些开销，例如创建会更慢，因为为这些表分配的 *.frm* 文件需要刷新到磁盘。可以通过禁止 sync_frm 选项来加速，但这可能会导致潜在的风险。

如果确实需要使用临时表，也应该在关闭备库前确保 Slave_open_temp_tables 状态变量值为 0。如果不是 0，在重启备库后就可能会出现问题。合适的流程是执行 STOP SLAVE，检查变量，然后再关闭备库。如果在停止复制前检查变量，可能会发生竞争条件的风险。

10.7.11 不复制所有的更新

如果错误地使用 SET SQL_LOG_BIN=0 或者没有理解过滤规则，备库可能会丢失主库上已经发生的更新。有时候希望利用此特性来做归档，但常常会导致意外并出现不好的结果。

例如，假设设置了 replicate_do_db 规则，把 sakila 数据库的数据复制到某一台备库上。如果在主库上执行如下语句，会导致主备数据不一致：

```
mysql> USE test;
mysql> UPDATE sakila.actor ...
```

其他类型的语句甚至会因为没有复制依赖导致备库复制抛出错误而失败。

10.7.12 InnoDB 加锁读引起的锁争用

正常情况下，InnoDB 的读操作是非阻塞的，但在某些情况下需要加锁。特别是在使用基于语句的复制方式时，执行 INSERT...SELECT 操作会锁定源表上的所有行。MySQL 需要加锁以确保该语句的执行结果在主库和备库上是一致的。实际上，加锁导致主库上的语句串行化，以确保和备库上执行的方式相符。

这种设计可能导致锁竞争、阻塞，以及锁等待超时等情况。一种缓解的办法就是避免让事务开启太久以减少阻塞。可以在主库上尽快地提交事务以释放锁。

把大命令拆分成小命令，使其尽可能简短。这也是一种减少锁竞争的有效方法。即使有
504 时很难做到，但也是值得的（使用 Percona Toolkit 中的 *pt-archiver* 工具会很简单）。

另一种方法是替换掉 INSERT...SELECT 语句，在主库上先执行 SELECT INTO OUTFILE，再执行 LOAD DATA INFILE。这种方法更快，并且不需要加锁。这种方法很特殊，但有时还是有用的。最大的问题是为输出文件选择一个唯一的名字，并在完成后清理掉文件。可以通过之前讨论过的 CONNECTION_ID() 来保证文件名的唯一性，并且可以使用定时任

务（UNIX 的 *crontab*，Windows 平台的计划任务）在连接不再使用这些文件后进行自动清理。

也可以尝试关闭上面的这种锁机制，而不是使用上面的变通方法。有一种方法可以做到，但在大多数场景下并不是好办法，备库可能会在不知不觉间就失去和主库的数据同步。这也会导致在做恢复时二进制日志变得毫无用处。但如果确实觉得这么做的利大于弊，可以使用下面的办法来关闭这种锁机制：

```
# THIS IS NOT SAFE!
innodb_locks_unsafe_for_binlog = 1
```

这使得查询的结果所依赖的数据不再加锁。如果第二条查询修改了数据并在第一条查询之前先提交。在主库和备库上执行这两条语句的结果可能不相同。对于复制和基于时间点的恢复都是如此。

为了了解锁定读取是如何防止混乱的，假设有两张表：一个没有数据，另一个只有一行数据，值为 99。有两个事务更新数据。事务 1 将第二张表的数据插入到第一张表，事务 2 更新第二张表（源表），如图 10-16 所示。

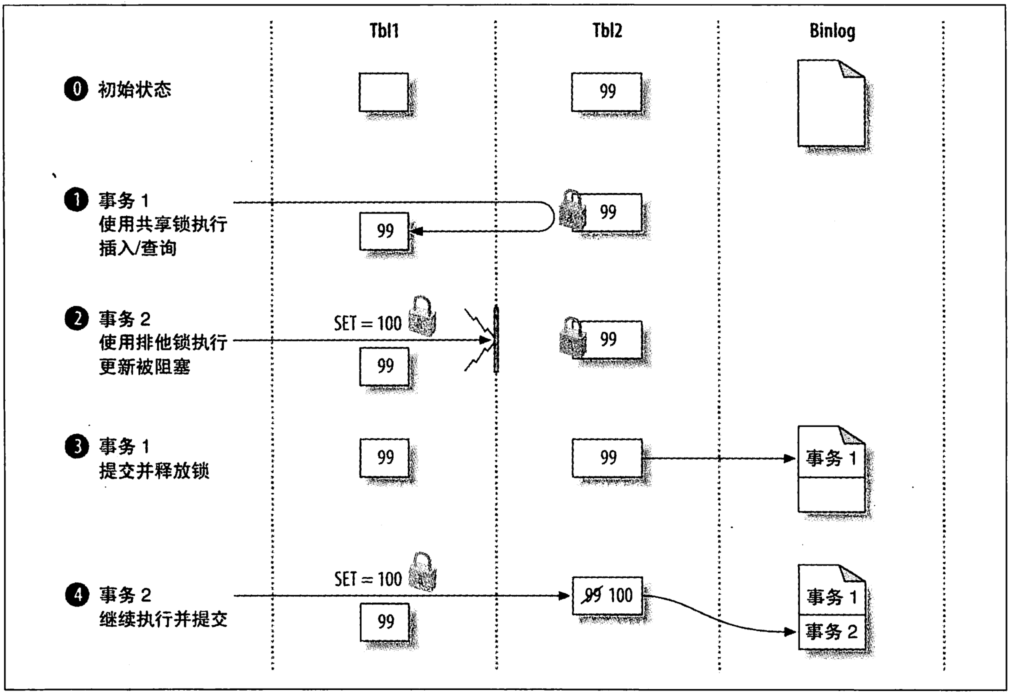


图10-16：两个事务更新数据，使用共享锁串行化更新

第二步非常重要，事务 2 尝试去更新源表，这需要在更新的行上加排他锁（写锁）。排他锁与其他锁是不相容的，包括事务 1 在行记录上加的共享锁。因此事务 2 需要等待直到事务 1 完成。事务按照其提交的顺序在二进制日志中记录，所以在备库重放这些事务时产生相同的结果。

但从另一方面来说，如果事务 1 没有在读取的行上加共享锁，就无法保证了。图 10-17 显示了在没有锁的情况下可能的事件序列。

如果没有加锁，记录在日志中的事务顺序在主备上可能会产生不同的结果。MySQL 会先记录事务 2，这会影响到事务 1 在备库上的结果，而主库上则不会发生，从而导致了主备的数据不一致。

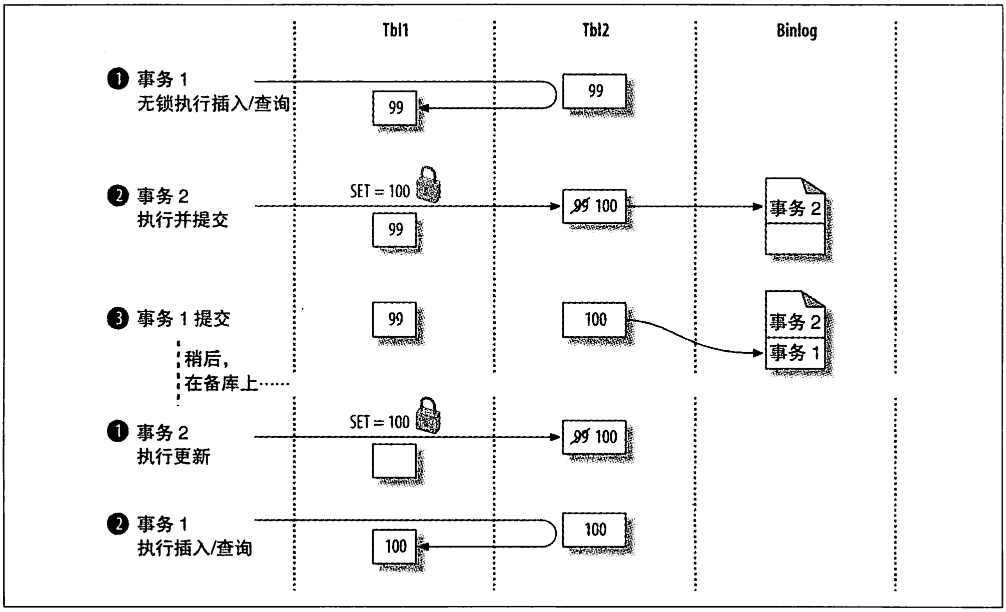


图10-17：两个事务更新数据，但未使用共享锁来串行化更新

505 我们强烈建议在大多数情况下将 `innodb_locks_unsafe_for_binlog` 的值设置为 0。基于行的复制由于记录了数据的变化而非语句，因此不会存在这个问题。

10.7.13 在主 – 主复制结构中写入两台主库

试图向两台主库写入并不是一个好主意。如果同时还希望安全地写入两台主库，会碰到很多问题，有些问题可以解决，有些则很难。一个专业人员可能需要经历大量的教训才能明白其中的不同。

在 MySQL 5.0 中，有两个变量可以用于帮助解决 AUTO_INCREMENT 自增主键冲突的问题：auto_increment_increment 和 auto_increment_offset。可以通过设置这两个变量来错开主库和备库生成的数字，这样可以避免自增列的冲突。

但是这并不能解决所有由于同时写入两台主库所带来的问题；自增问题只是其中的一小部分。而且这种做法也带来了一些新的问题：

- 很难在复制拓扑间做故障转移。 506
- 由于在数字之间出现间隙，会引起键空间的浪费。
- 只有在使用了 AUTO_INCREMENT 主键的时候才有用。有时候使用 AUTO_INCREMENT 列作为主键并不总是好主意。

你也可以自己来生成不冲突的主键值。一种办法是创建一个多个列的主键，第一列使用服务器 ID 值。这种办法很好，但却使得主键的值变得更大，会对 InnoDB 二级索引键值产生多重影响。

也可以使用只有一列的主键，在主键的高字节位存储服务器 ID。简单的左移位（除法）和加法就可以实现。例如，使用的是无符号 BIGINT（64 位）的高 8 位来保存服务器 ID，可以按照如下方法在服务器 15 上插入值 11：

```
mysql> INSERT INTO test(pk_col, ...) VALUES( (15 << 56) + 11, ...);
```

如果想把结果转换为二进制，并将其填充为 64 位，其效果显而易见：

```
mysql> SELECT LPAD(CONV(pk_col, 10, 2), 64, '0') FROM test;
+------------------------------------------------------------------+
| LPAD(CONV(pk_col, 10, 2), 64, '0')                               |
+------------------------------------------------------------------+
| 0000111100000000000000000000000000000000000000000000000000001011 |
+------------------------------------------------------------------+
```

该方法的缺点是需要额外的方式来产生键值，因为 AUTO_INCREMENT 无法做到这一点。不要在 INSERT 语句中将常量 15 替换为 @@server_id，因为这可能在备库产生不同的结果。 507

还可以使用 MD5() 或 UUID() 等函数来获取伪随机数，但这样做性能可能会很差，因为它们产生的值较大，并且本质上是随机的，这尤其会影响到 InnoDB（除非是在应用中产生值，否则不要使用 UUID()，因为基于语句的复制模式下 UUID() 不能正确复制）。

这个问题很难解决，我们通常推荐重构应用程序，以保证只有一个主库是可写的。谁能想得到呢？

10.7.14 过大的复制延迟

复制延迟是一个很普遍的问题。不管怎么样，最好在设计应用程序时能够让其容忍备库出现延迟。如果系统在备库出现延迟时就无法很好地工作，那么应用程序也许就不应该用到复制。但是也有一些办法可以让备库跟上主库。

MySQL 单线程复制的设计导致备库的效率相当低下。即使备库有很多磁盘、CPU 或者内存，也会很容易落后于主库。因为备库的单线程通常只会有效地使用一个 CPU 和磁盘。而事实上，备库通常都会和主库使用相同配置的机器。

备库上的锁同样也是问题。其他在备库运行的查询可能会阻塞住复制线程。因为复制是单线程的，复制线程在等待时将无法做别的事情。

复制一般有两种产生延迟的方式：突然产生延迟然后再跟上，或者稳定的延迟增大。前一种通常是由于一条运行很长时间的查询导致的，而后者即使在没有长时间运行的查询时也会出现。

不幸的是，目前我们没那么容易确定备库是否接近其容量上限。正如之前提到的。如果负载总是保持均匀的，备库在负载达到 99% 时和其负载在 10% 的时候表现的性能相同，但一旦达到 100% 时就会突然开始产生延迟。但实际上负载不太可能很稳定，所以当备库接近写容量时，就可能在尖峰负载时看到复制延迟的增加。

当备库无法跟上时，可以记录备库上的查询并使用一个日志分析工具找出哪里慢了。不要依赖于自己的直觉，也不要基于查询在主库上的查询性能进行判断，因为主库和备库性能特征很不相同。最好的分析办法是暂时在备库上打开慢查询日志记录，然后使用第
508 3 章讨论的 *pt-query-digest* 工具来分析。如果打开了 `log_slow_slave_statements` 选项，在标准的 MySQL 慢查询日志能够记录 MySQL 5.1 及更新的版本中复制线程执行的语句，这样就可以找到在复制时哪些语句执行慢了。Percona Server 和 MariaDB 允许开启或禁止该选项而无须重启服务器。

除了购买更快的磁盘和 CPU（固态硬盘能够提供极大的帮助，详细参阅第 9 章），备库没有太多的调优空间。大部分选项都是禁止某些额外的工作以减少备库的负载。一个简单的办法是配置 InnoDB，使其不要那么频繁地刷新磁盘，这样事务会提交得更快些。可以通过设置 `innodb_flush_log_at_trx_commit` 的值为 2 来实现。还可以在备库上禁止二进制日志记录，把 `innodb_locks_unsafe_for_binlog` 设置为 1，并把 MyISAM 的 `delay_key_write` 设置为 ALL。但是这些设置以牺牲安全换取速度。如果需要将备库提升为主库，记得把这些选项设置回安全的值。

不要重复写操作中代价较高的部分

重构应用程序并且/或者优化查询通常是最好的保持备库同步的办法。尝试去最小化系统中重复的工作。任何主库上昂贵的写操作都会在每一个备库上重放。如果可以把工作转移到备库，那么就只有一台备库需要执行，然后我们可以把写的结果回传到主库，例如，通过执行 LOAD DATA INFILE。

这里有个例子，假设有一个大表，需要汇总到一个小表中用于日常的操作：

```
mysql> REPLACE INTO main_db.summary_table (col1, col2, ...)
    -> SELECT col1, sum(col2, ...)
    -> FROM main_db.enormous_table GROUP BY col1;
```

如果在主库上执行查询，每个备库将同样需要执行庞大的 GROUP BY 查询。当进行太多这样的操作时，备库将无法跟上。把这些工作转移到一台备库上也许会有帮助。在备库上创建一个特别保留的数据库，用于避免和从主库上复制的数据产生冲突。可以执行以下查询：

```
mysql> REPLACE INTO summary_db.summary_table (col1, col2, ...)
    -> SELECT col1, sum(col2, ...)
    -> FROM main_db.enormous_table GROUP BY col1;
```

现在可以执行 SELECT INTO OUTFILE，然后再执行 LOAD DATA INFILE，将结果集加载到主库中。现在重复工作被简化为 LOAD DATA INFILE 操作。如果有 *N* 个备库，就节约了 *N*-1 次庞大的 GROUP BY 操作。

该策略的问题是需要处理陈旧数据。有时候从备库读取的数据和写入主库的数据很难保持一致（下一章我们会详细描述这个问题）。如果难以在备库上读取数据，依然能够简化并节省库备工作。如果分离查询的 REPLACE 和 SELECT 部分，就可以把结果返回给应用程序，然后将其插入到主库中。首先，在主库执行如下查询： 509

```
mysql> SELECT col1, sum(col2, ...) FROM main_db.enormous_table GROUP BY col1;
```

然后为结果集的每一行重复执行如下语句，将结果插入到汇总表中：

```
mysql> REPLACE INTO main_db.summary_table (col1, col2, ...) VALUES (?, ?, ...);
```

这种方法再次避免了在备库上执行查询中的 GROUP BY 部分。将 SELECT 和 REPLACE 分离后意味着查询的 SELECT 操作不会在每一台备库上重放。

这种通用的策略——节约了备库上昂贵的写入操作部分——在很多情况下很有帮助：计算查询的结果代价很昂贵，但一旦计算出来后，处理就很容易。

在复制之外并行写入

另一种避免备库严重延迟的办法是绕过复制。任何在主库的写入操作必须在备库串行化。因此有理由认为“串行化写入”不能充分利用资源。所有写操作都应该从主库传递到备库吗？如何把备库有限的串行写入容量留给那些真正需要通过复制进行的写入？

这种考虑有助于对写入进行区分。特别是，如果能确定一些写入可以轻易地在复制之外执行，就可以并行化这些操作以利用备库的写入容量。

一个很好的例子是之前讨论过的数据归档。OLTP 归档需求通常是简单的单行操作。如果只是把不需要的记录从一个表移到另一个表，就没有必要将这些写入复制到备库。可以禁止归档查询记录到二进制日志中，然后分别在主库和备库上单独执行这些归档查询。

自己复制数据到另外一台服务器，而不是通过复制，这听起来有些疯狂，但却对一些应用有意义，特别是如果应用是某些表的唯一更新源。复制的瓶颈通常集中在小部分表上。如果能在复制之外单独处理这些表，就能够显著地加快复制。

为复制线程预取缓存

如果有正确的工作负载，就能通过预先将数据读入内存中，以受益于在备库上的并行 I/O 所带来的好处。这种方式并不广为人知。大多数人不会使用，因为除非有正确的工作负载特性和硬件配置，否则可能没有任何用处。我们刚刚讨论过的其他几种变通方式通常是更好的选择，并且有更多的方法来应用它们。但是我们知道也有小部分应用会受益于数据预取。

510

有两种可行的实现方法。一种是通过程序实现，略微比备库 SQL 线程提前读取中继日志并将其转换为 SELECT 语句执行。这会使得服务器将数据从磁盘加载到内存中，这样当 SQL 线程执行到相应的语句时，就无须从磁盘读取数据。事实上，SELECT 语句可以并行地执行，所以可以加速 SQL 线程的串行 I/O。当一条语句正在执行时，下一条语句需要的数据也正在从磁盘加载到内存中。

如果满足下面这些条件，预取可能会有效：

- 复制 SQL 线程是 I/O 密集型的，但备库服务器并不是 I/O 密集型的。一个完全的 I/O 密集型服务器不会受益于预取，因为它没有多余的磁盘性能来提供预取。
- 备库有多个硬盘驱动器，也许 8 个或者更多。
- 使用的是 InnoDB 引擎，并且工作集远不能完全加载到内存中。

一个受益于预读取的例子是随机单行 UPDATE 语句，这些语句通常在主库上高并发执行。DELETE 语句也可能受益于这种方法，但 INSERT 语句则不太可能会——尤其是当顺序插

入时——因为前一次插入已经使索引“预热”了。

如果表上有很多索引，同样无法预取所有将要被修改的数据。UPDATE 语句可能需要更新所有索引，但 SELECT 语句通常只会读取主键和一个二级索引。UPDATE 语句依然需要去读取其他索引的数据以进行更新。在多索引表上这种方法的效率会降低。

这种技术并不是“银弹”，有很多原因会导致其不能工作，甚至适得其反。只有在清楚硬件和操作系统的状况时才能尝试这种方法。我们知道有些人利用这种办法将复制速度提升了 300% 到 400%，但我们也尝试过很多次，并发现这种方法常常无法工作。正确地设置参数非常重要，但并没有绝对正确的参数组合。

mk-slave-prefetch 是 Maatkit 中的一款工具，该工具实现了本节所提到的预取策略。*mk-slave-prefetch* 本身有很多复杂的策略以保证其在尽可能多的场景下工作。但缺点是它实在太复杂并且需要许多专业知识来使用。另一款工具是 Anders Karlsson 的 *slavereadahead* 工具，可以从 *http://sourceforge.net/projects/slavereadahead/* 获得。

另一种方法在写作本书时还正在开发中，它是在 InnoDB 内部实现的。它可以允许设置 511
事务为特殊的模式，以允许 InnoDB 执行“假”更新。因此可以使用一个程序来执行这些假更新，这样复制线程就可以更快地执行真正的更新。我们已经在 Percona Server 中为一个非常流行的互联网网络应用单独开发了该功能。可以去检查一下此特性现在的状态，因为在本书出版时或许已经更新过了。

如果正在考虑这项技术，可以从一个熟悉其工作原理及可用选项的专家那里获得很好的建议。这应该作为其他方案都不可行时最后的解决办法。

10.7.15 来自主库的过大的包

另一个难以追踪的问题是主库的 max_allowed_packet 值和备库的不匹配。在这种情况下，主库可能会记录一个备库认为过大的包。当备库获取到该二进制日志事件时，可能会碰到各种各样的问题，包括无限报错和重试，或者中继日志损坏。

10.7.16 受限制的复制带宽

如果使用受限的带宽进行复制，可以开启备库上的 slave_compressed_protocol 选项（在 MySQL 4.0 及新版本中可用）。当备库连接主库时，会请求一个被压缩的连接——和 MySQL 客户端使用的压缩连接一样。使用的压缩引擎是 *zlib*，我们的测试表明它能将文本类型的数据压缩到大约其原始大小的三分之一。其代价是需要额外的 CPU 时间，包括在主库上压缩数据和在备库上解压数据。

如果主库和其备库间的连接是慢速连接，可能需要将分发主库和备库分布在同一地点。这样就只有一台服务器通过慢速连接和主库相连，可以减少链路上的带宽负载以及主库的 CPU 负载。

10.7.17 磁盘空间不足

复制有可能因为二进制日志、中继日志或临时文件将磁盘撑满。特别是在主库上执行了 `LOAD DATA INFILE` 查询并在备库开启了 `log_slave_updates` 选项。延迟越严重，接收到但尚未执行的中继日志会占用越多的磁盘空间。可以通过监控磁盘并设置 `relay_log_space` 选项来避免这个问题。

512 10.7.18 复制的局限性

MySQL 复制可能失败或者不同步，不管有没有报错，这是因为其内部的限制导致的。大量的 SQL 函数和编程实践不能被可靠地复制（本章我们已经讨论了许多这样的例子）。很难确保应用代码里不会出现这样或那样的问题，特别是应用或者团队非常庞大的时候。[注 22]

另外一个问题是服务器的 Bug，虽然听起来很消极，但大多数 MySQL 的主版本都存在着历史遗留的复制 Bug。特别是每个主版本的第一个版本。诸如存储过程这样的新特性常常会导致更多的问题。

MySQL 复制非常复杂。应用程序越复杂，你就需要越小心。但是如果学会了如何使用，复制会工作得很好。

10.8 复制有多快

关于复制的一个比较普遍的问题是复制到底有多快？简单来讲，它与 MySQL 从主库复制事件并在备库重放的速度一样快。如果网络很慢并且二进制日志事件很大，记录二进制日志和在备库上执行的延迟可能会非常明显。如果查询需要执行很长时间而网络很快，通常可以认为查询时间占据了更多的复制时间开销。

更完整的答案是计算每一步花费的时间，并找到应用中耗时最多的那一部分。一些读者可能只关注主库上记录事件和将事件复制到中继日志的时间间隔。对于那些想了解更多细节的读者，我们可以做一个快速的实验。

我们在本书的第一版详细描述了复制的过程和 Giuseppe Maxia 提供的测量高精度复制速

注 22： 最近的 MySQL 版本没有 `forbid_operations_unsafe_for_replication` 选项，但它确实对一些不安全的事情起到了警示，甚至拒绝。

度的方法[注23]。我们创建了一个非确定性的用户自定义函数（UDF），以微秒精度返回系统时间（源代码参阅前面的“用户定义函数”）：

```
mysql> SELECT NOW_USEC()
+----------------------------+
| NOW_USEC()                 |
+----------------------------+
| 2007-10-23 10:41:10.743917 |
+----------------------------+
```

513

首先将 NOW_USEC() 函数的值插入到主库的一张表中，然后比较它在备库上的值，以此来测量复制的速度。

为了测量延迟，我们在一台服务器上开启两个 MySQL 实例，以避免由于时钟引起的不精确。我们将其中一个实例配置为另一个的备库，然后在主库实例上执行如下语句：

```
mysql> CREATE TABLE test.lag_test(
    ->   id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->   now_usec VARCHAR(26) NOT NULL
    -> );
mysql> INSERT INTO test.lag_test(now_usec) VALUES( NOW_USEC() );
```

我们使用的是 VARCHAR 列，因为 MySQL 内建的时间类型只能精确到秒（尽管一些时间函数可以执行小于秒级别的计算），剩下的就是比较主备的差异。这里我们使用 Federated 表[注24]。在备库上执行：

```
mysql> CREATE TABLE test.master_val (
    ->   id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->   now_usec VARCHAR(26) NOT NULL
    -> ) ENGINE=FEDERATED
    ->   CONNECTION='mysql://user:pass@127.0.0.1/test/lag_test',;
```

简单的关联和 TIMESTAMPDIFF() 函数可以微秒精度显示主库和备库上执行查询的延迟。

```
mysql> SELECT m.id, TIMESTAMPDIFF(FRAC_SECOND, m.now_usec, s.now_usec) AS usec_lag
    -> FROM test.lag_test as s
    ->    INNER JOIN test.master_val AS m USING(id);
+----+----------+
| id | usec_lag |
+----+----------+
|  1 |      476 |
+----+----------+
```

我们使用 Perl 脚本向主库中插入 1 000 行数据，每个插入间有 10 毫秒的延时，以避免主备实例竞争 CPU 时间。然后创建一个临时表来存储每个事件的延迟：

注 23： 查看 *http://datacharmer.blogspot.com/2006/04/measuring-replication-speed.html*。
注 24： 顺便说一下，这也是一些作者唯一一次使用 Federated 存储引擎。

```
mysql> CREATE TABLE test.lag AS
     > SELECT TIMESTAMPDIFF(FRAC_SECOND, m.now_usec, s.now_usec) AS lag
    -> FROM test.master_val AS m
    ->    INNER JOIN test.lag_test as s USING(id);
```

接着根据延迟时间分组，可以看到最常见的延迟时间是多少：

514

```
mysql> SELECT ROUND(lag / 1000000.0, 4) * 1000 AS msec_lag, COUNT(*)
    -> FROM lag
    -> GROUP BY msec_lag
    -> ORDER BY msec_lag;
+----------+----------+
| msec_lag | COUNT(*) |
+----------+----------+
|   0.1000 |      392 |
|   0.2000 |      468 |
|   0.3000 |       75 |
|   0.4000 |       32 |
|   0.5000 |       15 |
|   0.6000 |        9 |
|   0.7000 |        2 |
|   1.3000 |        2 |
|   1.4000 |        1 |
|   1.8000 |        1 |
|   4.6000 |        1 |
|   6.6000 |        1 |
|  24.3000 |        1 |
+----------+----------+
```

结果显示大多数小查询在主库上的执行时间和备库上的执行时间间隔大多数小于 0.3 毫秒。

复制过程中没有计算的部分是事件在主库上记录到二进制日志后需要多长时间传递到备库。有必要知道这一点，因为备库越快接收到日志事件越好。如果备库已经接收到了事件，它就能在主库崩溃时提供一个拷贝。

尽管我们的测量结果没有精确地显示这部分需要多长时间，但理论上非常快（例如，仅仅受限于网络速度）。MySQL 二进制日志转储线程并没有通过轮询的方式从主库请求事件，而是由主库来通知备库新的事件，因为前者低效且缓慢。从主库读取一个二进制日志事件是一个阻塞型网络调用，当主库记录事件后，马上就开始发送。因此可以说，只要复制线程被唤醒并且能够通过网络传输数据，事件就会很快到达备库。

10.9 MySQL 复制的高级特性

Oracle 对 MySQL 5.5 的复制有着明显的改进。更多的特性还在开发中，MySQL 5.6 将包含这些新特性。一些改进使得复制更加强健，例如，增加了多线程（并行）复制以减少当前单线程复制的瓶颈。另外，还有一些改进增加了一些高级特性，使得复制更加灵活

并可控制。我们不会描述太多尚未GA的功能，但会讨论一些MySQL 5.5关于复制的改进。

第一个是半同步复制，基于Google多年前所做的工作。这是自MySQL 5.1引入行复制后最大的改进。它可以帮助你确保备库拥有主库数据的拷贝，减少了潜在的数据丢失危险。 515

半同步复制在提交过程中增加了一个延迟：当提交事务时，在客户端接收到查询结束反馈前必须保证二进制日志已经传输到至少一台备库上。主库将事务提交到磁盘上之后会增加一些延迟。同样的，这也增加了客户端的延迟，因此其执行大量事务的速度不会比将这些事务传递给备库的速度更快。

关于半同步，有一些普遍的误解，下面是它不会去做的：

- 在备库提示其已经收到事件前，会阻塞主库上的事务提交。事实上在主库上已经完成事务提交，只有通知客户端被延迟了。
- 直到备库执行完事务后，才不会阻塞客户端。备库在接收到事务后发送反馈而非完成事务后发送。
- 半同步不总是能够工作。如果备库一直没有回应已收到事件，会超时并转化为正常的异步复制模式。

尽管如此，这仍然是一个很好用的工具，有助于确保备库提供更好的冗余度和持久性。

在性能方面，从客户端的角度来看，增加了事务提交的延时，延时的多少取决于网络传输，数据写入和刷新到备库磁盘的时间（如果开启了配置）以及备库反馈的网络时间。听起来似乎这是累加的，但测试证明这些几乎是不重要的，也许延迟是由其他原因引起的。Giuseppe Maxia发现每次提交大约延时200微秒[注25]。对于小事务开销可能会比较明显，这也是预期中的。

事实上半同步复制在某些场景下确实能够提供足够的灵活性以改善性能，在主库关闭`sync_binlog`的情况下保证更加安全。写入远程的内存（一台备库反馈）比写入本地的磁盘（写入并刷新）要更快。Henrik Ingo运行了一些性能测试表明，使用半同步复制相比在主库上进行强持久化的性能有两倍的改善[注26]。在任何系统上都没有绝对的持久化——只有更加高的持久化层次——并且看起来半同步复制应该是一种比其他替代方案开销更小的系统数据持久化方法。 516

除了半同步复制，MySQL 5.5还提供了复制心跳，保证备库一直与主库相联系，避免悄无声息地断开连接。如果出现断开的网络连接，备库会注意到丢失的心跳数据。当使用

注25： 参阅 *http://datacharmer.blogspot.com/2011/05/price-of-safe-data-benchmarking-semi.html*。
注26： 参阅 *http://openlife.cc/blogs/2011/may/drbd-and-semi-sync-shootout-large-server*。

基于行的复制时，还提供了一种改进的能力来处理主库和备库上不同的数据类型。有几个选项可以用于配置复制元数据文件是如何刷新到磁盘以及在一次崩溃后如何处理中继日志，减少了备库崩溃恢复后出现问题的概率。

我们还没有看到 MySQL 5.5 对复制的改进大规模地在生产环境进行部署，因此还需要进行更多的研究。

除了上面提到的，这里简要地列出其他一些改进，包括 MySQL 以及第三方分支，例如 Percona Server 以及 MariaDB：

- Oracle 在 MySQL 5.6 实验室版本和开发里程碑版本中有许多的改进。
  - 事务复制状态，即使崩溃也不会导致元数据失去同步（Percona Server 和 MariaDB 已经以别的形式实现了）。
  - 二进制日志的 checksum 值，用于检测中继日志中损坏的事件。
  - 备库延迟复制，用于替代 Percona Toolkit 中的 *pt-slave-delay* 工具。
  - 允许基于行的二进制日志事件也包含在主库执行的 SQL。
  - 实现多线程复制（并行复制）。
- MySQL5.6、Percona Server、Facebook 以及 MariaDB 提供了三种修复方法解决了 MySQL 5.0 引入的 GROUP COMMIT 的问题。

10.10 其他复制技术

MySQL 内建的复制并不是将数据从一台服务器复制到另外一台服务器的唯一办法，尽管大多数时候是最好的办法。（与 PostgreSQL 相比，MySQL 并没有大量附加的复制选项，可能是因为复制功能在早期就已经引入了）。

我们已经讨论了 MySQL 复制的一些扩展技术，如 Oracle GoldenGate，但对大多数工具我们都不熟悉，因此无法讨论太多。但是有两个我们需要指出来，第一个是 Percona XtraDB Cluster 的同步复制，我们会在第 12 章介绍，因为它比较适合在高可用性这一章讲述。另一个是 Continuent 的 Tungsten Replicator（*http://code.google.com/p/tungsten-replicator/*）。

517 Tungsten 是一个用 Java 编写的开源的中间件复制产品。它的功能和 Oracle GoldenGate 类似，并且看起来在未来发布的版本中将逐步增加许多复杂的特性。在写作本书时，它已经提供了一些特性，例如，在服务器间复制数据、自动数据分片、在备库并发执行更新（多线程复制）、当主库失败时提升备库、跨平台复制，以及多源复制（多个复制源到一个目标）。它是 Tungsten 数据库 clustering suite 的开源版本。

Tungsten 同样实现了多主库集群，可以把写入指向集群中任意一台服务器。这种架构的实现通常都包含冲突发现与解决。这一点很难做到，并且不总是需要的。Tungsten 的实现稍微做了点限制，不是所有的数据都能在所有的节点写入，每个节点被标记为记录系统，以接收特定的数据。例如，在西雅图的办公室可以拥有并写入它的数据，然后复制到休斯敦和巴尔的摩。在休斯敦和巴尔的摩本地可以实现低延迟读数据，但在这里 Tungsten 不允许写入数据，这样数据冲突就不存在了。当然休斯敦和巴尔的摩可以更新它们自己的数据，并被复制到其他地点。这种“记录系统”方案解决了人们需要在环形结构中频繁调整内建 MySQL 复制的问题。我们之前讨论的环形复制还远远不够安全或强健。

Tungsten Replicator 不仅仅是嵌入或管理 MySQL 复制，而是直接替代它。它通过读取主库的二进制日志来获得数据更新，那里正是内建 MySQL 复制工作结束的地方，然后由 Tungsten Replicator 接管。它读取二进制日志，并抽取出事务，然后在备库执行它们。

该过程比 MySQL 复制本身有更丰富的功能集。实际上，Tungsten Replicator 是第一个提供 MySQL 并行复制支持的。虽然我们还没有看到其被应用到生产环境中，但它声称能够提供最多三倍的复制速度改善，具体取决于负载特性。基于该架构以及我们对该产品的了解，这看起来是可信的。

以下是关于 Tungsten Replicator 中值得欣赏的部分 ：

- 它提供了内建的数据一致性检查。
- 提供了插件特性，因此你可以编写自己的函数。MySQL 的复制源代码非常难以理解并且很难去修改。即使非常聪明的程序员在试图修改时，也会引入新的 Bug。因而能有种途径去修改复制而无须修改 MySQL 的复制代码，是非常理想的。
- 拥有全局事务 ID，能够帮助你了解每个服务器相互之间的状态而无须去匹配二进制日志名和偏移量。
- 它是一个高可用的解决方案，能够快速地将一台备库提升为主库。 518
- 提供异构数据复制(例如,在 MySQL 和 PostgreSQL 之间或者 MySQL 和 Oracle 之间)。
- 支持不同版本的 MySQL 复制，以防止 MySQL 复制不能反向兼容。这对某些升级的场景非常有用。当升级运行得不理想时，你可能无法设计一个可行的回滚方案，或者必须升级服务器到一个并不是你期望的版本。
- 并行复制的设计非常适用于共享应用程序或多任务应用程序。
- Java 应用能够明确地写入主库并从备库读取。
- 得益于 Giuseppe Maxia 作为 QA 主管的大量工作，现在比以往更加简单并且更加容易配置和管理。

以下是它的一些缺点：

- 它比内建的 MySQL 复制更加复杂，有更多可变动的地方需要配置和管理，毕竟它是一个中间件。
- 在你的应用栈中需要多学习和理解一个新的工具。
- 它并不像内建的 MySQL 复制那样轻量级，并且没有同样的性能。使用 Tungsten Replicator 进行单线程复制比 MySQL 的单线程复制要慢。
- 作为 MySQL 复制并没有经过广泛的测试和部署，所以 Bug 和问题的风险很高。

总而言之，我们很高兴 Tungsten Replicator 是可用的，并且在积极的开发中，稳定地释放新的特性和功能。拥有一个可替代内建 MySQL 复制的选择，这非常棒，使得 MySQL 能够适用于更多的应用场景，并且足够灵活，能够满足内建的 MySQL 复制可能永远无法满足的需求。

10.11 总结

MySQL 复制是其内建功能中的"瑞士军刀"，显著增加了 MySQL 的功能和可用性。事实上这也是 MySQL 这么快就如此流行的关键原因之一。

尽管复制有许多限制和风险，但大多数相对不重要或者对大多数用户而言是可以避免的。许多缺点只在一些高级特性的特殊行为中，这些特性对少数需要的人而言是有帮助的，但大多数人并不会用到。

519 正因为复制提供了如此重要和复杂的功能，服务器本身不提供所有其他你需要的功能，例如，配置、监控、管理和优化。第三方工具可以很好地帮助你。虽然可能有失偏颇，但我们认为最值得关注的工具一定是 Percona Toolkit 和 Percona XtraBackup，它们能够很好地改进你对复制的使用。在使用别的工具前，建议你先检查它们的测试集合，如果没有正式的、自动化的测试集合，在将其应用到你的数据之前请认真考虑。

对于复制，应该铭记 K.I.S.S[注27] 原则。不要按照想象做事，例如，使用环形复制、黑洞表或者复制过滤，除非确实有需要。使用复制简单地去镜像一份完整的数据拷贝，包括所有的权限。在各方面保持你的主备库相同可以帮助你避免很多问题。

谈到保持主库和备库相同，这里有一个简短但很重要的列表告诉你在使用复制的时候需要做什么：

- 使用 Percona Toolkit 中的 *pt-table-checksum* 以确定备库是主库的真实拷贝。
- 监控复制以确定其正在运行并且没有落后于主库。

注 27： Keep It Simple, Schwartz! 总之一些人认为这是 K.I.S.S 的含义。

- 理解复制的异步本质，并且设计你的应用以避免或容忍从备库读取脏的数据。
- 在一个复制拓扑中不要写入超过一个服务器，把备库配置为只读，并降低权限以阻止对数据的改变。
- 打开本章所讨论的那些明智并且安全的设置。

正如我们将要在第 12 章讨论的，复制失败是 MySQL 故障时间中最普遍的原因之一。为了避免复制的问题，阅读第 12 章，并尝试应用其给予的建议。你同样也应该通读 MySQL 手册中关于复制的章节，并了解复制如何工作以及如何去管理它。如果乐于阅读，Charles Bell et al. 所著的 *MySQL High Availability*（O'Reilly）一书中有许多关于复制内部的有用信息。但你依然需要阅读手册！

第 11 章 521

可扩展的MySQL

本章将展示如何构建一个基于 MySQL 的应用，并且当规模变得越来越庞大时，还能保证快速、高效并且经济。

有些应用仅仅适用于一台或少数几台服务器，那么哪些可扩展性建议是和这些应用相关的呢？大多数人从不会维护超大规模的系统，并且通常也无法效仿在主流大公司所使用的策略。本章会涵盖这一系列的策略。我们已经建立或者协助建立了许多应用，包括从单台或少量服务器的应用到使用上千台服务器的应用。选择一个合适的策略能够大大地节约时间和金钱。

MySQL 经常被批评很难进行扩展，有些情况下这种看法是正确的，但如果选择正确的架构并很好地实现，就能够非常好地扩展 MySQL。但是扩展性并不是一个很好理解的主题，所以我们先来理清一些容易混淆的地方。

11.1 什么是可扩展性

人们常常把诸如“可扩展性”、“高可用性”以及“性能”等术语在一些非正式的场合用作同义词，但事实上它们是完全不同的。在第 3 章已经解释过，我们将性能定义为响应时间。我们也可以很精确地定义可扩展性，稍后将完整讨论。简要地说，可扩展性表明了当需要增加资源以执行更多工作时系统能够获得划算的等同提升（equal bang for the buck）的能力。缺乏扩展能力的系统在达到收益递减的转折点后，将无法进一步增长。

容量是一个和可扩展性相关的概念。系统容量表示在一定时间内能够完成的工作量[注1]，
但容量必须是可以有效利用的。系统的最大吞吐量并不等同于容量。大多数基准测试能 522

注 1： 从物理学来看，单位时间内做的功称为功率（power），而在计算机领域，“power”是一个被反复使用的术语，含义模糊，因此应避免使用它。但是关于容量的精确定义是系统的最大功率输出。

够衡量一个系统的最大吞吐量，但真实的系统一般不会使用到极限。如果达到最大吞吐量，则性能会下降，并且响应时间会变得不可接受地大且非常不稳定。我们将系统的真实容量定义为在保证可接受的性能的情况下能够达到的吞吐量。这就是为什么基准测试的结果通常不应该简化为一个单独的数字。

容量和可扩展性并不依赖于性能。以高速公路上的汽车来类比的话：

- 性能是汽车的时速。
- 容量是车道数乘以最大安全时速。
- 可扩展性就是在不减慢交通的情况下，能增加更多车和车道的程度。

在这个类比中，可扩展性依赖于多个条件：换道设计得是否合理、路上有多少车抛锚或者发生事故，汽车行驶速度是否不同或者是否频繁变换车道——但一般来说和汽车的引擎是否强大无关。这并不是说性能不重要，性能确实重要，只是需要指出，即使系统性能不是很高也可以具备可扩展性。

从较高层次看，可扩展性就是能够通过增加资源来提升容量的能力。

即使 MySQL 架构是可扩展的，但应用本身也可能无法扩展，如果很难增加容量，不管原因是什么，应用都是不可扩展的。之前我们从吞吐量方面来定义容量，但同样也需要从较高的层次来看待容量问题。从有利的角度来看，容量可以简单地认为是处理负载的能力，从不同的角度来考虑负载很有帮助。

数据量

应用所能累积的数据量是可扩展性最普遍的挑战，特别是对于现在的许多互联网应用而言，这些应用从不删除任何数据。例如社交网站，通常从不会删除老的消息或评论。

用户量

即使每个用户只有少量的数据，但在累计到一定数量的用户后，数据量也会开始不成比例地增长且速度快过用户数增长。更多的用户意味着要处理更多的事务，并且事务数可能和用户数不成比例。最后，大量用户（以及更多的数据）也意味着更多复杂的查询，特别是查询跟用户关系相关时（用户间的关联数可以用 $N\times(N-1)$ 来计算，这里 N 表示用户数）。

用户活跃度

不是所有的用户活跃度都相同，并且用户活跃度也不总是不变的。如果用户突然变得活跃，例如由于增加了一个吸引人的新特性，那么负载可能会明显提升。用户活

跃度不仅仅指页面浏览数，即使同样的页面浏览数，如果网站的某个需要执行大量工作的部分变得流行，也可能导致更多的工作。另外，某些用户也会比其他用户更活跃：他们可能比一般人有更多的朋友、消息和照片。 523

相关数据集的大小

如果用户间存在关系，应用可能需要在整个相关联用户群体上执行查询和计算，这比处理一个一个的用户和用户数据要复杂得多。社交网站经常会遇到由那些人气很旺的用户组或朋友很多的用户所带来的挑战[注2]。

11.1.1 正式的可扩展性定义

有必要探讨一下可扩展性在数学上的定义了，这有助于在更高层次的概念上清晰地理解可扩展性。如果没有这样的基础，就可能无法理解或精确地表达可扩展性。不过不用担心，这里不会涉及高等数学，即使不是数学天才，也能够很直观地理解它。

关键是之前我们使用的短语："划算的等同提升（equal bang for the buck）"。另一种说法是，可扩展性是当增加资源以处理负载和增加容量时系统能够获得的投资产出率（ROI）。假设有一个只有一台服务器的系统，并且能够测量它的最大容量，如图 11-1 所示。

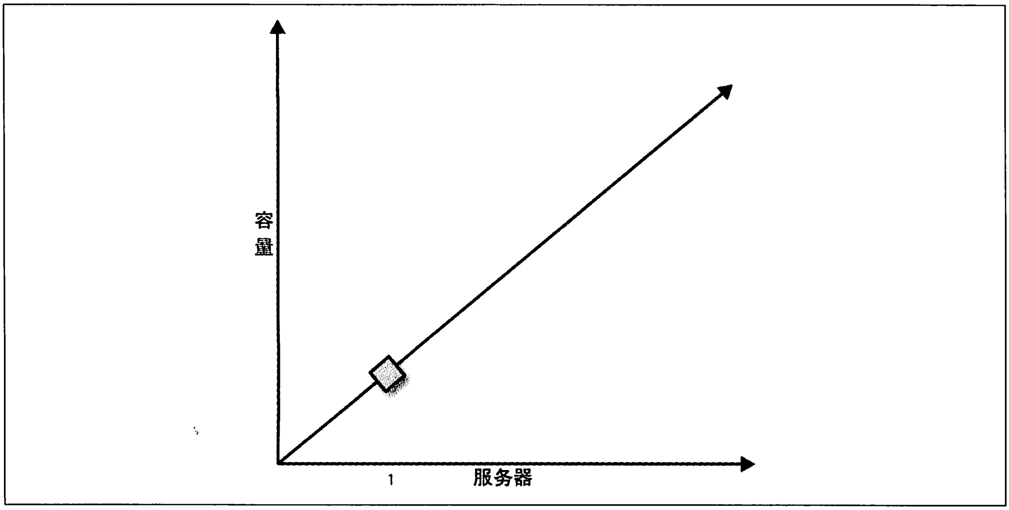


图11-1：一个只有一台服务器的系统

假设现在我们增加一台服务器，系统的能力加倍，如图 11-2 所示。 524

注 2： Justin Bieber，我们仍然爱你！

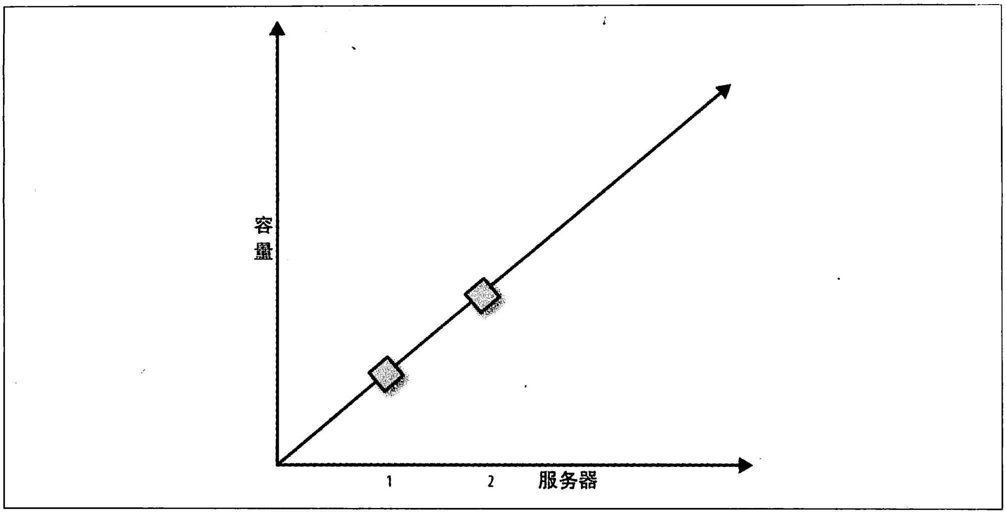


图11-2：一个线性扩展的系统能由两台服务器获得两倍容量

这就是线性扩展。我们增加了一倍的服务器，结果增加了一倍的容量。大部分系统并不是线性扩展的，而是如图 11-3 所示的扩展方式。

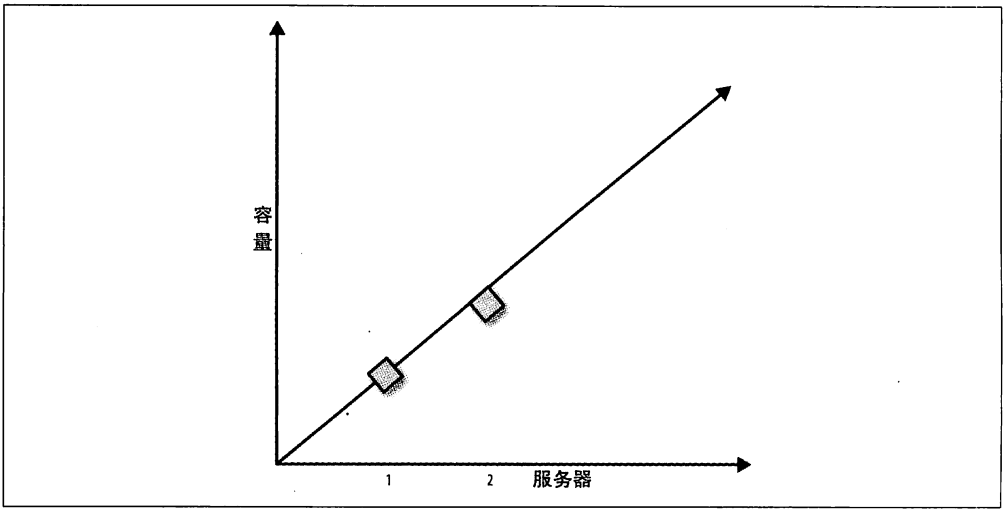


图11-3：一个非线性扩展的系统

525 大部分系统都只能以比线性扩展略低的扩展系数进行扩展。越高的扩展系数会导致越大的线性偏差。事实上，多数系统最终会达到一个最大吞吐量临界点，超过这个点后增加

投入反而会带来负回报——继续增加更多工作负载，实际上会降低系统的吞吐量。[注3]

这怎么可能呢？这些年产生了许多可扩展性模型，它们有着不同程度的良好表现和实用性。我们这里所讲的可扩展性模型是基于某些能够影响系统扩展的内在机制。这就是 Neil J. Gunther 博士提出的通用可扩展性定律（Universal Scalability Law，USL）。Gunther 博士将这些详尽地写到了他的书中，包括 *Guerrilla Capacity Planning*（Springer）。这里我们不会深入到背后的数学理论中，如果你对此感兴趣，他撰写的书籍以及由他的公司 Performance Dynamics 提供的训练课程可能是比较好的资源。[注4]

简而言之，USL 说的是线性扩展的偏差可通过两个因素来建立模型：无法并发执行的一部分工作，以及需要交互的另外一部分工作。为第一个因素建模就有了著名的 Amdahl 定律，它会导致吞吐量趋于平缓。如果部分任务无法并行，那么不管你如何分而治之，该任务至少需要串行部分的时间。

增加第二个因素——内部节点间或者进程间的通信——到 Amdahl 定律就得出了 USL。这种通信的代价取决于通信信道的数量，而信道的数量将按照系统内工作者数量的二次方增长。因此最终开销比带来的收益增长得更快，这是产生扩展性倒退的原因。图 11-4 阐明了目前讨论到的三个概念：线性扩展、Amdahl 扩展，以及 USL 扩展。大多数真实系统看起来更像 USL 曲线。

USL 可以应用于硬件和软件领域。对于硬件，横轴表示硬件的数量，例如服务器数量或 CPU 数量。每个硬件的工作量、数据大小以及查询的复杂度必须保持为常量[注5]。对于软件，横轴表示并发度，例如用户数或线程数。每个并发的工作量必须保持为常量。

有一点很重要，USL 并不能完美地描述真实系统，它只是一个简化模型。但这是一个很好的框架，可用于理解为什么系统增长无法带来等同的收益。它也揭示了一个构建高可扩展性系统的重要原则：在系统内尽量避免串行化和交互。

可以衡量一个系统并使用回归来确定串行和交互的量。你可以将它作为容量规划和性能预测评估的最优上限值。也可以检查系统是怎么偏离 USL 模型的，将其作为最差下限值以指出系统的哪一部分没有表现出它应有的性能。这两种情况下，USL 给出了一个讨论可扩展性的参考。如果没有 USL，那即使盯着系统看也无法知道期望的结果是什么。如

注 3：事实上，“投资产出率”也可以从金融投资的角度来考虑。将一个组件的容量升级到两倍所需要付出的常常不止是最初开销的两倍。虽然在现实世界里我们常常这么考虑，但在讨论中会将其忽略掉，因为它会使一个已经复杂的主题变得更加复杂。

注 4：你也可以阅读我们的白皮书“*Forecasting MySQL Scalability with the Universal Scalability Law*”，该书扼要地总结了 USL 中的数学运算和法则，可以从 *http://www.percona.com* 获得。

注 5：现实中很难精确定义硬件的可扩展性，因为当你改变你的系统中的服务器数量时很难保证那些变量不变。

果想深入了解这个主题，最好去看一下对应的书籍。Gunther 博士已经写得很清楚，因此我们不会再深入讨论下去。

526

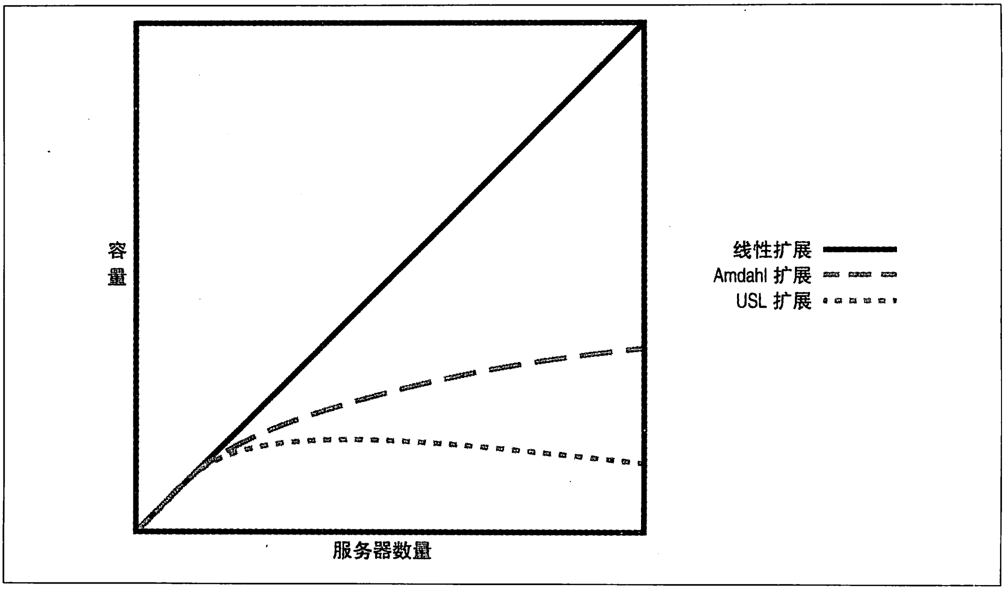


图11-4：线性扩展、Amdahl扩展以及USL扩展定律

527 另外一个理解可扩展性问题的框架是约束理论，它解释了如何通过减少依赖事件和统计变化（statistical variation）来改进系统的吞吐量和性能。这在 Eliyahu M. Goldratt 所撰写的 *The Goal*（North River）一书中有描述，其中有一个关于管理制造业设备的延伸的比喻。尽管这看起来和数据库服务器没有什么关联，但其中包含的法则和排队理论以及其他运筹学方面是一样的。

扩展模型不是最终定论

虽然有许多理论，但在现实中能做到何种程度呢？正如牛顿定律被证明只有远低于光速时才合理，那些“扩展性定律”也只是在某些场景下才能很好工作的简化模型。有一种说法认为所有的模型都是错误的，但有一些模型还是有用的，特别是 USL 能够帮助理解一些导致扩展性差的因素。

当工作负载和其所运行的系统存在微妙的关系时，USL 理论可能失效。例如，一个 USL 无法很好建模的常见情况是：当集群的总内存由于数据集大小而发生改变时，也会导致系统的行为发生变化。USL 不允许比线性更好的可扩展性，但现实中可能会发生这样的事情：增加系统的资源后，原来一部分 I/O 密集型的工作变成了纯内存工作，因此获得了超过线性的性能扩展。

还有一些情况，USL 无法很好描述系统行为。当系统或数据集大小改变时算法的复杂度可能改变，类似这样的情况下可能就无法建立模型（USL 由 $O(1)$ 复杂度和 $O(N^2)$ 复杂度两部分构成，那么对于诸如 $O(\log N)$ 或者 $O(N\log N)$ 这样复杂度的部分呢？）。根据一些思考和实际经验，我们可以将 USL 扩展以覆盖这些比较普遍的场景中的一部分。但这会将一个简单并且有用的模型变得复杂并难以使用。事实上，它在很多情况下都是很好的，足以为你所能想象到的系统行为建立模型。这也是为什么我们发现它是在正确性和有效性之间的一个很好的妥协。

简单地说：有保留地使用模型，并且在使用中验证你的发现。

11.2 扩展 MySQL

如果将应用所有的数据简单地放到单个 MySQL 服务器实例上，则无法很好地扩展，迟早会碰到性能瓶颈。对于许多类型的应用，传统的解决方法是购买更多强悍的机器，也就是常说的“垂直扩展”或者“向上扩展”。另外一个与之相反的方法是将任务分配到多台计算机上，这通常被称为“水平扩展”或者“向外扩展”。我们将讨论如何联合使用向上扩展和向外扩展的解决方案，以及如何使用集群方案来进行扩展。最后，大部分应用还会有一些很少或者从不需要的数据，这些数据可以被清理或归档。我们将这个方案称为“向内扩展”，这么取名是为了和其他策略相匹配。

11.2.1 规划可扩展性

人们通常只有在无法满足增加的负载时才会考虑到可扩展性，具体表现为工作负载从 CPU 密集型变成 I/O 密集型，并发查询的竞争，以及不断增大的延迟。主要原因是查询 528
的复杂度增加或者内存中驻留着一部分不再使用的数据或者索引。你可能看到一部分类型的查询发生改变，例如大的查询或者复杂查询常常比那些小的查询更影响系统。

如果是可扩展的应用，则可以简单地增加更多的服务器来分担负载，这样就没有性能问题了。但如果不是可扩展的，你会发现自己将遭遇到无穷无尽的问题。可以通过规划可扩展性来避免这个问题。

规划可扩展性最困难的部分是估算需要承担的负载到底有多少。这个值不一定非常精确，但必须在一定的数量级范围内。如果估计过高，会浪费开发资源。但如果低估了，则难以应付可能的负载。

另外还需要大致正确地估计日程表——也就是说，需要知道底线在哪里。对于一些应用，一个简单的原型可以很好地工作几个月，从而有时间去筹资建立一个更加可扩展的架构。对于其他的一些应用，你可能需要当前的架构能够为未来两年提供足够的容量。

以下问题可以帮助规划可扩展性：

- 应用的功能完成了多少？许多建议的可扩展性解决方案可能会导致实现某些功能变得更加困难。如果应用的某些核心功能还没有开始实现，就很难看出如何在一个可扩展的应用中实现它们。同样地，在知道这些特性如何真实地工作之前也很难决定使用哪一种可扩展性解决方案。
- 预期的最大负载是多少？应用应当在最大负载下也可以正常工作。如果你的网站和 Yahoo! News 或者 Slashdot 的首页一样，会发生什么呢？即使不是很热门的网站，也同样有最高负载。比如，对于一个在线零售商，假日期间——尤其是在圣诞前的几个星期——通常是负载达到巅峰的时候。在美国，情人节和母亲节前的周末对于在线花店来说也是负载高峰期。
- 如果依赖系统的每个部分来分担负载，在某个部分失效时会发生什么呢？例如，如果依赖备库来分担读负载，当其中一个失效时，是否还能正常处理请求？是否需要禁用一些功能？可以预先准备一些空闲容量来防范这种问题。

11.2.2 为扩展赢得时间

在理想情况下，应该是计划先行、拥有足够的开发者、有花不完的预算，等等。但现实中这些情况会很复杂，在扩展应用时常常需要做一些妥协，特别是需要把对系统大的改
529 动推迟一段时间再执行。在深入 MySQL 扩展的细节前，以下是一些可以做的准备工作：

优化性能

很多时候可以通过一个简单的改动来获得明显的性能提升，例如为表建立正确的索引或从 MyISAM 切换到 InnoDB 存储引擎。如果遇到了性能限制，可以打开查询日志进行分析，详情请参阅第 3 章。

在修复了大多数主要的问题后，会到达一个收益递减点，这时候提升性能会变得越来越困难。每个新的优化都可能耗费更多的精力但只有很小的提升，并会使应用更加复杂。

购买性能更强的硬件

升级或增加服务器在某些场景下行之有效，特别是对处于软件生命周期早期的应用，

购买更多的服务器或者增加内存通常是个好办法。另一个选择是尽量在一台服务器上运行应用程序。比起修改应用的设计，购买更多的硬件可能是更实际的办法，特别是时间紧急并且缺乏开发者的时候。

如果应用很小或者被设计为便于利用更多的硬件，那么购买更多的硬件应该是行之有效的办法。对于新应用这是很普遍的，因为它们通常很小或者设计合理。但对于大型的旧应用，购买更多硬件可能没什么效果，或者代价太高。服务器从 1 台增加到 3 台或许算不了什么，但从 100 台增加到 300 台就是另外一回事了——代价非常昂贵。如果是这样，花一些时间和精力来尽可能地提升现有系统的性能就很划算。

11.2.3 向上扩展

向上扩展（有时也称为垂直扩展）意味着购买更多性能强悍的硬件，对很多应用来说这是唯一需要做的事情。这种策略有很多好处。例如，单台服务器比多台服务器更加容易维护和开发，能显著节约开销。在单台服务器上备份和恢复应用同样很简单，因为无须关心一致性或者哪个数据集是权威的。当然，还有一些别的原因。从复杂性的成本来说，向上扩展比向外扩展更简单。

向上扩展的空间其实也很大。拥有 0.5TB 内存、32 核（或者更多）CPU 以及更强悍 I/O 性能的（例如 PCIe 卡的 flash 存储）商用服务器现在很容易获得。优秀的应用和数据库设计，以及很好的性能优化技能，可以帮助你在这样的服务器上建立一个 MySQL 大型 530
应用。

在现代硬件上 MySQL 能扩展到多大的规模呢？尽管可以在非常强大的服务器上运行，但和大多数数据库服务器一样，在增加硬件资源的情况下 MySQL 也无法很好地扩展（非常奇怪！）。为了更好地在大型服务器上运行 MySQL，一定要尽量选择最新的版本。由于内部可扩展性问题，MySQL 5.0 和 5.1 在大型硬件里的表现并不理想。建议使用 MySQL 5.5 或者更新的版本，或者 Percona Server 5.1 及后续版本。即便如此，当前合理的“收益递减点”的机器配置大约是 256GB RAM，32 核 CPU 以及一个 PCIe flash 驱动器。如果继续提升硬件的配置，MySQL 的性能虽然还能有所提升，但性价比就会降低，实际上，在更强大的系统上，也可以通过运行多个小的 MySQL 实例来替代单个大实例，这样可以获得更好的性能。当然，机器配置的变化速度非常快，这个建议也许很快就会过时了。

向上扩展的策略能够顶一段时间，实际很多应用是不会达到天花板的。但是如果应用变得非常庞大[注6]，向上扩展可能就没有办法了。第一个原因是钱，无论服务器上运行什么

注 6： 我们避免使用措辞“web 扩展”（web scale），因为它已经变得毫无意义，参阅 *http://www.xtranormal.com/watch/6995033/*。

样的软件，从某种角度来看，向上扩展都是个糟糕的财务决策，当超出硬件能够提供的最优性价比时，就会需要非同寻常的特殊配置的硬件，这样的硬件往往非常昂贵。这意味着能向上扩展到什么地步是有实际的限制的。如果使用了复制，那么当主库升级到高端硬件后，一般是不太可能配置出一台能够跟上主库的强大备库的。一个高负载的主库通常可以承担比拥有同样配置的备库更多的工作，因为备库的复制线程无法高效地利用多核 CPU 和磁盘资源。

最后，向上扩展不是无限制的，即使最强大的计算机也有限制。单服务器应用通常会首先达到读限制，特别是执行复杂的读查询时。类似这样的查询在 MySQL 内部是单线程的，因此只能使用一个 CPU，这种情况下花钱也无法提升多少性能。即使购买最快的 CPU 也仅仅会是商用 CPU 的几倍速度。增加更多的 CPU 或 CPU 核数并不能使慢查询执行得更快。当数据变得庞大以至于无法有效缓存时，内存也会成为瓶颈，这通常表现为很高的磁盘使用率，而磁盘是现代计算机中最慢的部分。

无法使用向上扩展最明显的场景是云计算。在大多数公有云中都无法获得性能非常强的服务器，如果应用肯定会变得非常庞大，就不能选择向上扩展的方式。在第 13 章我们会深入这个话题。

531 因此，我们建议，如果系统确实有可能碰到可扩展性的天花板，并且会导致严重的业务问题，那就不要无限制地做向上扩展的规划。如果你知道应用会变得很庞大，在实现另外一种解决方案前，短期内购买更优的服务器是可以的。但是最终还是需要向外扩展，这也是下一节我们要讲述的主题。

11.2.4 向外扩展

可以把向外扩展（有时也称为横向扩展或者水平扩展）策略划分为三个部分：复制、拆分，以及数据分片（sharding）。

最简单也最常见的向外扩展的方法是通过复制将数据分发到多个服务器上，然后将备库用于读查询。这种技术对于以读为主的应用很有效。它也有一些缺点，例如重复缓存，但如果数据规模有限这就不是问题。关于这些问题我们在前一章已经讨论得足够多，后面会继续提到。

另外一个比较常见的向外扩展方法是将工作负载分布到多个“节点”。具体如何分布工作负载是一个复杂的话题。许多大型的 MySQL 应用不能自动分布负载，就算有也没有做到完全的自动化。本节我们会讨论一些可能的分布负载的方案，并探讨它们的优点和缺点。

在 MySQL 架构中，一个节点（node）就是一个功能部件。如果没有规划冗余和高可用性，

那么一个节点可能就是一台服务器。如果设计的是能够故障转移的冗余系统，那么一个节点通常可能是下面的某一种：

- 一个主—主复制双机结构，拥有一个主动服务器和被动服务器。
- 一个主库和多个备库。
- 一个主动服务器，并使用分布式复制块设备（DRBD）作为备用服务器。
- 一个基于存储区域网络（SAN）的"集群"。

大多数情况下，一个节点内的所有服务器应该拥有相同的数据。我们倾向于把主—主复制架构作为两台服务器的主动—被动节点。

1. 按功能拆分

按功能拆分，或者说按职责拆分，意味着不同的节点执行不同的任务。我们之前已经提到了一些类似的实现，在前一章我们描述了如何为 OLTP 和 OLAP 工作负载设计不同的服务器。按功能拆分采取的策略比这些更进一步，将独立的服务器或节点分配给不同的应用，这样每个节点只包含它的特定应用所需要的数据。

这里我们显式地使用了"应用"一词。所指的并不是一个单独的计算机程序，而是相关的一系列程序，这些程序可以很容易地彼此分离，没有关联。例如，如果有一个网站，各个部分无须共享数据，那么可以按照网站的功能区域进行划分。门户网站常常把不同的栏目放在一起；在门户网站，可以浏览网站新闻、论坛，寻求支持和访问知识库，等等。这些不同功能区域的数据可以放到专用的 MySQL 服务器中，如图 11-5 所示。 532

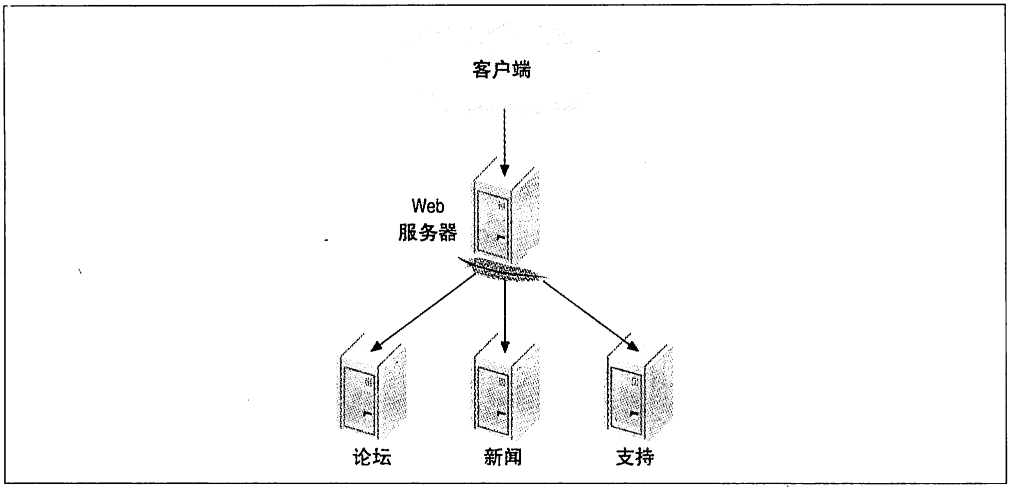


图11-5：一个门户网站以及专用于不同功能区域的节点

如果应用很庞大，每个功能区域还可以拥有其专用的 Web 服务器，但没有专用的数据库服务器这么常见。

另一个可能的按功能划分方法是对单个服务器的数据进行划分，并确保划分的表集合之间不会执行关联操作。当必须执行关联操作时，如果对性能要求不高，可以在应用中做关联。虽然有一些变通的方法，但它们有一个共同点，就是每种类型的数据只能在单个节点上找到。这并不是一种通用的分布数据方法，因为很难做到高效，并且相比其他方案没有任何优势。

归根结底，还是不能通过功能划分来无限地进行扩展，因为如果一个功能区域被捆绑到单个 MySQL 节点，就只能进行垂直扩展。其中的一个应用或者功能区域最终增长到非常庞大时，都会迫使你去寻求一个不同的策略。如果进行了太多的功能划分，以后就很难采用更具扩展性的设计了。

533

2. 数据分片

在目前用于扩展大型 MySQL 应用的方案中，数据分片[注7]是最通用且最成功的方法。它把数据分割成一小片，或者说一块，然后存储到不同的节点中。

数据分片在和某些类型的按功能划分联合使用时非常有用。大多数分片系统也有一些“全局的”数据不会被分片（例如城市列表或者登录数据）。全局数据一般存储在单个节点上，并且通常保存在类似 *memcached* 这样的缓存里。

事实上，大多数应用只会对需要的数据做分片——通常是那些将会增长得非常庞大的数据。假设正在构建的博客服务，预计会有 1000 万用户，这时候就无须对注册用户进行分片，因为完全可以将所有的用户（或者其中的活跃用户）放到内存中。假如用户数达到 5 亿，那么就可能需要对用户数据分片。用户产生的内容，例如发表的文章和评论，几乎肯定需要进行数据分片，因为这些数据非常庞大，并且还会越来越多。

大型应用可能有多个逻辑数据集，并且处理方式也可以各不相同。可以将它们存储到不同的服务器组上，但这并不是必需的。还可以以多种方式对数据进行分片，这取决于如何使用它们。下文我们会举例说明。

分片技术和大多数应用的最初设计有着显著的差异，并且很难将应用从单一数据存储转换为分片架构。如果在应用设计初期就已经预计到分片，那实现起来就容易得多。

许多一开始没有建立分片架构的应用都会碰到规模扩大的情形。例如，可以使用复制来扩展博客服务的读查询，直到它不再奏效。然后可以把服务器划分为三个部分：用户信息、

注 7： 分片也被称为“分裂”、“分区”，但是我们使用“分片”以避免混淆。谷歌将它称为“分片”，如果谷歌觉得这样称呼合适，我们采取这种称呼也就合适了。

文章，以及评论。可以将这些数据放到不同的服务器上（按功能划分），也许可以使用面向服务的架构，并在应用层执行联合查询。图 11-6 显示了从单台服务器到按功能划分的演变。

534

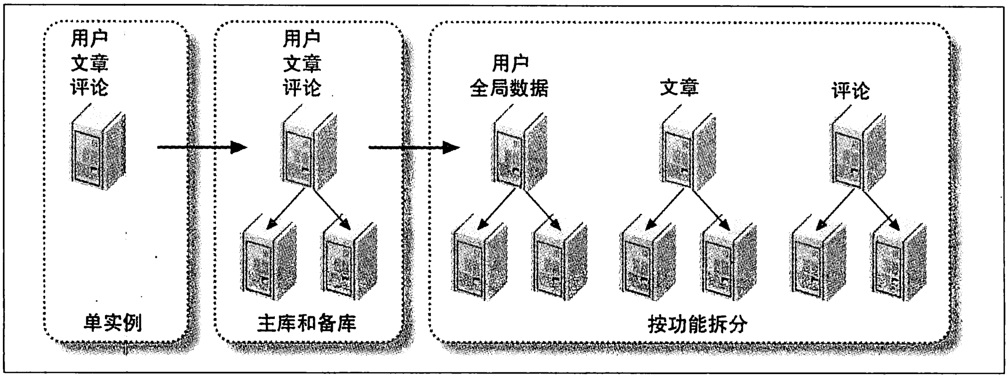


图11-6：从单个实例到按功能划分的数据存储

最后，可以通过用户 ID 来对文章和评论进行分片，而将用户信息保留在单个节点上。如果为全局节点配置一个主—备结构并为分片节点使用主—主结构，最终的数据存储可能如图 11-7 所示。

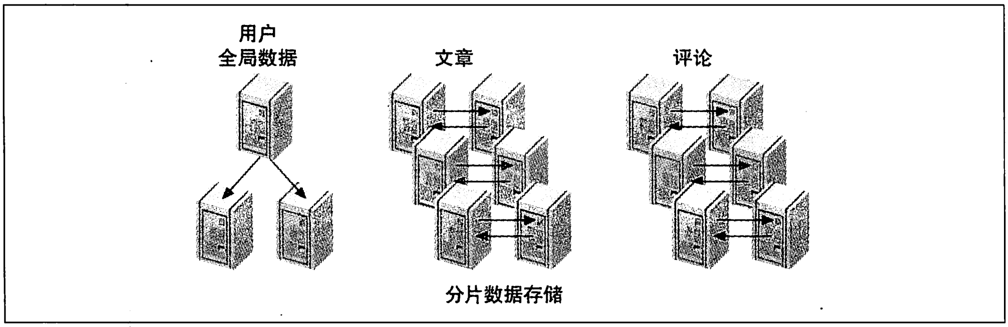


图11-7：一个全局节点和六个主—主结构节点的数据存储方式

如果事先知道应用会扩大到很大的规模，并且清楚按功能划分的局限性，就可以跳过中间步骤，直接从单个节点升级为分片数据存储。事实上，这种前瞻性可以帮你避免由于粗糙的分片方案带来的挑战。

采用分片的应用常会有一个数据库访问抽象层，用以降低应用和分片数据存储之间通信的复杂度，但无法完全隐藏分片。因为相比数据存储，应用通常更了解跟查询相关的一

些信息。太多的抽象会导致低效率，例如查询所有的节点，可实际上需要的数据只在单一节点上。

分片数据存储看起来像是优雅的解决方案，但很难实现。那为什么要选择这个架构呢？答案很简单：如果想扩展写容量，就必须切分数据。如果只有单台主库，那么不管有多少备库，写容量都是无法扩展的。对于上述缺点而言，数据分片是我们首选的解决方案。

535

分片？还是不分片？

这是一个问题，对吧？答案很简单：如非必要，尽量不分片。首先看是否能通过性能调优或者更好的应用或数据库设计来推迟分片。如果能足够长时间地推迟分片，也许可以直接购买更大的服务器，升级 MySQL 到性能更优的版本，然后继续使用单台服务器，也可以增加或减少复制。

简单的说，对单台服务器而言，数据大小或写负载变得太大时，分片将是不可避免的。如果不分片，而是尽可能地优化应用，系统能扩展到什么程度呢？答案可能会让你很惊讶。有些非常受欢迎的应用，你可能以为从一开始就分片了，但实际上直到已经值数十亿美元并且流量极其巨大也还没有采用分片的设计。分片不是城里唯一的游戏，在没有必要的情况下采用分片的架构来构建应用会步履维艰。

3．选择分区键（partitioning key）

数据分片最大的挑战是查找和获取数据：如何查找数据取决于如何进行分片。有很多方法，其中有一些方法会比另外一些更好。

我们的目标是对那些最重要并且频繁查询的数据减少分片（记住，可扩展性法则的其中一条就是要避免不同节点间的交互）。这其中最重要的是如何为数据选择一个或多个分区键。分区键决定了每一行分配到哪一个分片中。如果知道一个对象的分区键，就可以回答如下两个问题：

- 应该在哪里存储数据？
- 应该从哪里取到希望得到的数据？

后面将展示多个选择和使用分区键的方法。先看一个例子。假设像 MySQL NDB Cluster 那样来操作，并对每个表的主键使用哈希来将数据分割到各个分片中。这是一种非常简单的实现，但可扩展性不好，因为可能需要频繁检查所有分片来获得需要的数据。例如，如果想查看 user3 的博客文章，可以从哪里找到呢？由于使用主键值而非用户名进行分

割，博客文章可能均匀分散在所有的数据分片中。使用主键值哈希简化了判断数据存储在何处的操作，但却可能增加获取数据的难度，具体取决于需要什么数据以及是否知道主键。

跨多个分片的查询比单个分片上的查询性能要差，但只要不涉及太多的分片，也不会太糟糕。最糟糕的情况是不知道需要的数据存储在哪里，这时候就需要扫描所有分片。

一个好的分区键常常是数据库中一个非常重要的实体的主键。这些键值决定了分片单元。例如，如果通过用户 ID 或客户端 ID 来分割数据，分片单元就是用户或者客户端。

确定分区键一个比较好的办法是用实体—关系图，或一个等效的能显示所有实体及其关系的工具来展示数据模型。尽量把相关联的实体靠得更近。这样可以很直观地找出候选分区键。当然不要仅仅看图，同样也要考虑应用的查询。即使两个实体在某些方面是相关联的，但如果很少或几乎不对其做关联操作，也可以打断这种联系来实现分片。 536

某些数据模型比其他的更容易进行分片，具体取决于实体—关系图中的关联性程度。图 11-8 的左边展示了一个易于分片的数据模型，右边的那个则很难分片。

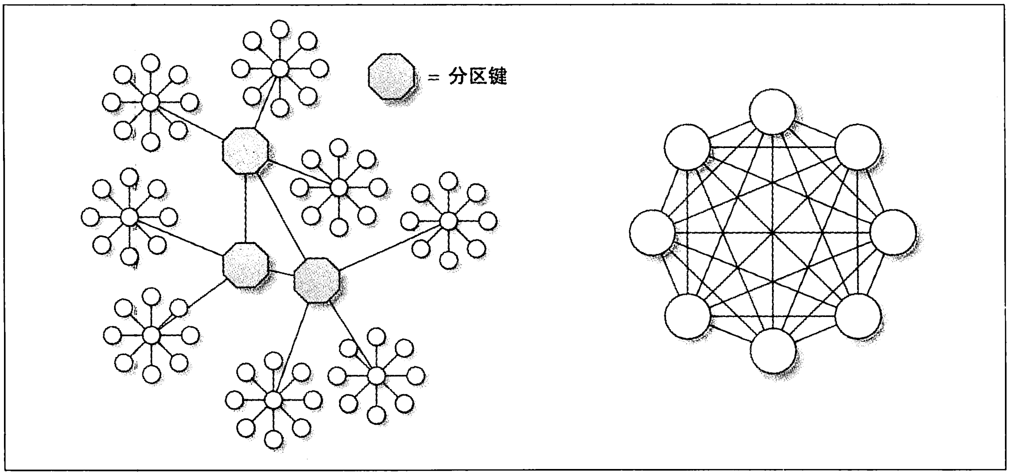


图11-8：两个数据模型，一个易于分片，另一个则难以分片

左边的数据模型比较容易分片，因为与之相连的子图中大多数节点只有一个连接，很容易切断子图之间的联系。右边的数据模型则很难分片，因为它没有类似的子图。幸好大多数数据模型更像左边的图。

选择分区键的时候，尽可能选择那些能够避免跨分片查询的，但同时也要让分片足够小，以免过大的数据片导致问题。如果可能，应该期望分片尽可能同样小，这样在为不同数

量的分片进行分组时能够很容易平衡。例如，如果应用只在美国使用，并且希望将数据集分割为 20 个分片，则可能不应该按照州来划分，因为加利福尼亚的人口非常多。但
537 可以按照县或者电话区号来划分，因为尽管并不是均匀分布的，但足以选择 20 个集合以粗略地表示等同的密集程度，并且基本上避免跨分片查询。

4. 多个分区键

复杂的数据模型会使数据分片更加困难。许多应用拥有多个分区键，特别是存在两个或更多个“维度”的时候。换句话说，应用需要从不同的角度看到有效且连贯的数据视图。这意味着某些数据在系统内至少需要存储两份。

例如，需要将博客应用的数据按照用户 ID 和文章 ID 进行分片，因为这两者都是应用查询数据时使用比较普遍的方式。试想一下这种情形：频繁地读取某个用户的所有文章，以及某个文章的所有评论。如果按用户分片就无法找到某篇文章的所有评论，而按文章分片则无法找到某个用户的所有文章。如果希望这两个查询都落到同一个分片上，就需要从两个维度进行分片。

需要多个分区键并不意味着需要去设计两个完全冗余的数据存储。我们来看看另一个例子：一个社交网站下的读书俱乐部站点，该站点的所有用户都可以对书进行评论。该网站可以显示所有书籍的所有评论，也能显示某个用户已经读过或评论过的所有书籍。

假设为用户数据和书籍数据都设计了分片数据存储。而评论同时拥有用户 ID 和评论 ID，这样就跨越了两个分片的边界。实际上却无须冗余存储两份评论数据，替代方案是，将评论和用户数据一起存储，然后把每个评论的标题和 ID 与书籍数据存储在一起。这样在渲染大多数关于某本书的评论的视图时无须同时访问用户和书籍数据存储，如果需要显示完整的评论内容，可以从用户数据存储中获得。

5. 跨分片查询

大多数分片应用多少都有一些查询需要对多个分片的数据进行聚合或关联操作。例如，一个读书俱乐部网站要显示最受欢迎或最活跃的用户，就必须访问每一个分片。如何让这类查询很好地执行，是实现数据分片的架构中最困难的部分。虽然从应用的角度来看，这是一条查询，但实际上需要拆分成多条并行执行的查询，每个分片上执行一条。一个设计良好的数据库抽象层能够减轻这个问题，但类似的查询仍然会比分片内查询要慢并且更加昂贵，所以通常会更加依赖缓存。

538 一些语言，如 PHP，对并行执行多条查询的支持不够好。普遍的做法是使用 C 或 Java 编写一个辅助应用来执行查询并聚合结果集。PHP 应用只需要查询该辅助应用即可，例如 Web 服务或者类似 Gearman 的工作者服务。

跨分片查询也可以借助汇总表来执行。可以遍历所有分片来生成汇总表并将结果在每个分片上冗余存储。如果在每个分片上存储重复数据太过浪费，也可以把汇总表放到另外一个数据存储中，这样就只需要存储一份了。

未分片的数据通常存储在全局节点中，可以使用缓存来分担负载。

如果数据的均衡分布非常重要，或者没有很好的分区键，一些应用会采用随机分片的方式。分布式检索应用就是个很好的例子。这种场景下，跨分片查询和聚合查询非常常见。

跨分片查询并不是数据分片面临的唯一难题。维护数据一致性同样困难。外键无法在分片间工作，因此需要由应用来检查参照一致性，或者只在分片内使用外键，因为分片内的内部一致性可能是最重要的。还可以使用 XA 事务，但由于开销太大，现实中使用很少。

还可以设计一些定期执行的清理过程。例如，如果一个用户的读书俱乐部账号到期，并不需要立刻将其移除。可以写一个定期任务将用户评论从每个书籍分片中移除。也可以写一个检查脚本周期性运行以确保分片间的数据一致性。

6. 分配数据、分片和节点

分片和节点不一定是一对一的关系，应该尽可能地让分片的大小比节点容量小很多，这样就可以在单个节点上存储多个分片。

保持分片足够小更容易管理。这将使数据的备份和恢复更加容易，如果表很小，那么像更改表结构这样的操作会更加容易。例如，假设有一个 100GB 的表，你可以直接存储，也可以将其划分为 100 个 1GB 的分片，并存储在单个节点上。现在假如要向表上增加一个索引，在单个 100GB 的表上的执行时间会比 100 个 1GB 分片上执行的总时间更长，因为 1GB 的分片更容易全部加载到内存中。并且在执行 ALTER TABLE 时还会导致数据不可用，阻塞 1GB 的数据比阻塞 100GB 的数据要好得多。

小一点的分片也便于转移。这有助于重新分配容量，平衡各个节点的分片。转移分片的效率一般都不高。通常需要先将受影响的分片设置为只读模式（这也是需要在应用中构建的特性），提取数据，然后转移到另外一个节点。这包括使用 *mysqldump* 获取数据然后使用 *mysql* 命令将其重新导入。如果使用的是 Percona Server，可以通过 XtraBackup 在服务器间转移文件，这比转储和重新载入要高效得多。 539

除了在节点间移动分片，你可能还需要考虑在分片间移动数据，并尽量不中断整个应用提供服务。如果分片太大，就很难通过移动整个分片来平衡容量，这时候可能需要将一部分数据（例如一个用户）转移到其他分片。分片间转移数据比转移分片要更复杂，应该尽量避免这么做。这也是我们建议设置分片大小尽量易于管理的原因之一。

分片的相对大小取决于应用的需求。简单的说，我们说的“易于管理的大小”是指保持表足够小，以便能在 5 或 10 分钟内提供日常的维护工作，例如 ALTER TABLE、CHECK TABLE 或者 OPTIMIZE TABLE。

如果将分片设置得太小，会产生太多的表，这可能引发文件系统或 MySQL 内部结构的问题。另外太小的分片还会导致跨分片查询增多。

7. 在节点上部署分片

需要确定如何在节点上部署数据分片。以下是一些常用的办法：

- 每个分片使用单一数据库，并且数据库名要相同。典型的应用场景是需要每个分片都能镜像到原应用的结构。这在部署多个应用实例，并且每个实例对应一个分片时很有用。
- 将多个分片的表放到一个数据库中，在每个表名上包含分片号（例如 bookclub.comments_23）。这种配置下，单个数据库可以支持多个数据分片。
- 为每个分片使用一个数据库，并在数据库中包含所有应用需要的表。在数据库名中包含分片号（例如表名可能是 bookclub_23.comments 或者 bookclub_23.users 等），但表名不包括分片号。当应用连接到单个数据库并且不在查询中指定数据库名时，这种做法很常见。其优点是无须为每个分片专门编写查询，也便于对只使用单个数据库的应用进行分片。
- 540 每个分片使用一个数据库，并在数据库名和表名中包含分片号（例如表名可以是 bookclub_23.comments_23）。
- 在每个节点上运行多个 MySQL 实例，每个实例上有一个或多个分片，可以使用上面提到的方式的任意组合来安排分片。

如果在表名中包含了分片号，就需要在查询模板里插入分片号。常用的方法是在查询中使用特殊的“神奇的”占位符，例如 sprintf() 这样的格式化函数中的 %s，或者使用变量做字符串插值。以下是在 PHP 中创建查询模板的方法：

```
$sql = "SELECT book_id, book_title FROM bookclub_%d.comments_%d... ";
$res = mysql_query(sprintf($sql, $shardno, $shardno), $conn);
```

也可以就使用字符串插值的方法：

```
$sql = "SELECT book_id, book_title FROM bookclub_$shardno.comments_$shardno ...";
$res = mysql_query($sql, $conn);
```

这在新应用中很容易实现，但对于已有的应用则有点困难。构建新应用时，查询模板并不是问题，我们倾向于使用每个分片一个数据库的方式，并把分片号写到数据库名和表

名中。这会增加例如 ALTER TABLE 这类操作的复杂度，但也有如下一些优点：

- 如果分片全部在一个数据库中，转移分片会比较容易。
- 因为数据库本身是文件系统中的一个目录，所以可以很方便地管理一个分片的文件。
- 如果分片互不关联，则很容易查看分片的大小。
- 全局唯一表名可避免误操作。如果表名每个地方都相同，很容易因为连接到错误的节点而查询了错误的分片，或者是将一个分片的数据误导入另外一个分片的表中。

你可能想知道应用的数据是否具有某种“分片亲和性”。也许将某些分片放在一起（在同一台服务器，同一个子网，同一个数据中心，或者同一个交换网络中）可以利用数据访问模式的相关性，能够带来些好处。例如，可以按照用户进行分片，然后将同一个国家的用户放到同一个节点的分片上。

为已有的应用增加分片支持的结果往往是一个节点对应一个分片。这种简化的设计可以减少对应用查询的修改。分片对应用而言通常是一种颠覆性的改变，所以应尽可能简化它。如果在分片后，每个节点看起来就像是整个应用数据的缩略图，就无须去改变大多数查询或担心查询是否传递到期望的节点。

8. 固定分配 541

将数据分配到分片中有两种主要的方法：固定分配和动态分配。两种方法都需要一个分区函数，使用行的分区键值作为输入，返回存储该行的分片。[注8]

固定分配使用的分区函数仅仅依赖于分区键的值。哈希函数和取模运算就是很好的例子。这些函数按照每个分区键的值将数据分散到一定数量的“桶”中。

假设有 100 个桶，你希望弄清楚用户 111 该放到哪个桶里。如果使用的是对数字求模的方式，答案很简单：111 对 100 取模的值为 11，所以应该将其放到第 11 个分片中。

而如果使用 CRC32() 函数来做哈希，答案是 81。

```
mysql> SELECT CRC32(111) % 100;
+------------------+
| CRC32(111) % 100 |
+------------------+
|               81 |
+------------------+
```

固定分配的主要优点是简单，开销低，甚至可以在应用中直接硬编码。

注 8： 这里的“函数”使用了其数学涵义，表示从输入（域）到输出（区间）的映射。如你所见，可以用很多方式来创建类似的函数，包括在数据库中使用查找表。

但固定分配也有如下缺点：

- 如果分片很大并且数量不多，就很难平衡不同分片间的负载。
- 固定分片的方式无法自定义数据放到哪个分片上，这一点对于那些在分片间负载不均衡的应用来说尤其重要。一些数据可能比其他的更加活跃，如果这些热点数据都分配到同一个分片中，固定分配的方式就无法通过热点数据转移的方式来平衡负载。（如果每个分片的数据量切分得比较小，这个问题就没那么严重，根据大数定律，这样做会更容易将热点数据平均分配到不同分片。）
- 修改分片策略通常比较困难，因为需要重新分配已有的数据。例如，如果通过模 10 的哈希函数来进行分片，就会有 10 个分片。如果应用增长使得分片变大，如果要拆分成 20 个分片，就需要对所有数据重新哈希，这会导致更新大量数据，并在分片间转移数据。

正是由于这些限制，我们倾向于为新应用选择动态分配的方式。但如果是为已有的应用
542 做分片，使用固定分配策略可能会更容易些，因为它更简单。也就是说，大多数使用固定分配的应用最后迟早要使用动态分配策略。

9. 动态分配

另外一个选择是使用动态分配，将每个数据单元映射到一个分片。假设一个有两列的表，包括用户 ID 和分片 ID。

```
CREATE TABLE user_to_shard (
   user_id INT NOT NULL,
   shard_id INT NOT NULL,
   PRIMARY KEY (user_id)
);
```

这个表本身就是分区函数。给定分区键（用户 ID）的值就可以获得分片号。如果该行不存在，就从目标分片中找到并将其加入到表中。也可以推迟更新——这就是动态分配的含义。

动态分配增加了分区函数的开销，因为需要额外调用一次外部资源，例如目录服务器（存储映射关系的数据存储节点）。出于效率方面的考虑，这种架构常常需要更多的分层。例如，可以使用一个分布式缓存系统将目录服务器的数据加载到内存中，因为这些数据平时改动很小。或者更普遍地，你可以直接向 USERS 表中增加一个 shard_id 列用于存储分片号。

动态分配的最大好处是可以对数据存储位置做细粒度的控制。这使得均衡分配数据到分片更加容易，并可提供适应未知改变的灵活性。

动态映射可以在简单的键—分片（key-to-shard）映射的基础上建立多层次的分片策略。例如，可以建立一个双重映射，将每个分片单元指定到一个分组中（例如，读书俱乐部的用户组），然后尽可能将这些组保持在同一个分片中。这样可以利用分片亲和性，避免跨分片查询。

如果使用动态分配策略，可以生成不均衡的分片。如果服务器能力不相同，或者希望将其中一些分片用于特定目的（例如归档数据），这可能会有用。如果能够做到随时重新平衡分片，也可以为分片和节点间维持一一对应的映射关系，这不会浪费容量。也有些人喜欢简单的每个节点一个分片的方式。（但是请记住，保持分片尽可能小是有好处的。）

动态分配以及灵活地利用分片亲和性有助于减轻规模扩大而带来的跨分片查询问题。假
设一个跨分片查询涉及四个节点，当使用固定分配时，任何给定的查询可能需要访问所
有分片，但动态分配策略则可能只需要在其中的三个节点上运行同样的查询。这看起来 543
没什么大区别，但考虑一下当数据存储增加到 400 个分片时会发生什么？固定分配策略
需要访问 400 个分片，而动态分配方式依然只需要访问 3 个。

动态分配可以让分片策略根据需要变得很复杂。固定分配则没有这么多选择。

10．混合动态分配和固定分配

可以混合使用固定分配和动态分配。这种方法通常很有用，有时候甚至必须要混合使用。目录映射不太大时，动态分配可以很好胜任。但如果分片单元太多，效果就会变差。

以一个存储网站链接的系统为例。这样一个站点需要存储数百亿的行，所使用的分区键是源地址和目的地址 URL 的组合。（这两个 URL 的任意一个都可能有好几亿的链接，因此，单独一个 URL 并不适合做分区键）。但是在映射表中存储所有的源地址和目的地址 URL 组合并不合理，因为数据量太大了，每个 URL 都需要很多存储空间。

一个解决方案是将 URL 相连并将其哈希到固定数目的桶中，然后把桶动态地映射到分片上。如果桶的数目足够大——例如 100 万个——你就能把大多数数据分配到每个分片上，获得动态分配的大部分好处，而无须使用庞大的映射表。

11．显式分配

第三种分配策略是在应用插入新的数据行时，显式地选择目标分片。这种策略在已有的数据上很难做到。所以在为应用增加分片时很少使用。但在某些情况下还是有用的。

这个方法是把数据分片号编码到 ID 中，这和之前提到的避免主—主复制主键冲突策略比较相似。（详情请参阅“在主—主复制结构中写入两台主库”。）

例如，假设应用要创建一个用户 3，将其分配到第 11 个分片中，并使用 BIGINT 列的高八位来保存分片号。这样最终的 ID 就是 (11<<56)+3，即 792633534417207299。应用可以很方便地从中抽取出用户 ID 和分片号，如下例所示。

```
mysql> SELECT (792633534417207299 >> 56) AS shard_id,
    -> 792633534417207299 & ~(11 << 56) AS user_id;
+----------+---------+
| shard_id | user_id |
+----------+---------+
|       11 |       3 |
+----------+---------+
```

544 现在假设要为该用户创建一条评论，并存储在同一个分片中。应用可以为该用户分配一个评论 ID 5，然后以同样的方式组合 5 和分片号 11。

这种方法的好处是每个对象的 ID 同时包含了分区键，而其他方法通常需要一次关联或查找来确定分区键。如果要从数据库中检索某个特定的评论，无须知道哪个用户拥有它；对象 ID 会告诉你到哪里去找。如果对象是通过用户 ID 动态分片的，就得先找到该评论的用户，然后通过目录服务器找到对应的数据分片。

另一个解决方案是将分区键存储在一个单独的列里。例如，你可能从不会单独引用评论 5，但是评论 5 属于用户 3。这种方法可能会让一些人高兴，因为这不违背第一范式；然而额外的列会增加开销、编码，以及其他不便之处。（这也是我们将两值存在单独一列的优点之一。）

显式分配的缺点是分片方式是固定的，很难做到分片间的负载均衡。但结合固定分配和动态分配，该方法就能够很好地工作。不再像之前那样哈希到固定数目的桶里并将其映射到节点，而是将桶作为对象的一部分进行编码。这样应用就能够控制数据的存储位置，因此可以将相关联的数据一起放到同样的分片中。

BoardReader（*http://boardreader.com*）使用了该技术的一个变种：它把分区键编码到 Sphinx 的文档 ID 内。这使得在分片数据存储中查找每个查询结果的关联数据变得容易，更多关于 Sphinx 的内容可以查阅附录 F。

我们讨论了混合分配方式，因为在某些场景下它是有用的。但正常情况下我们并不推荐这样用。我们倾向于尽可能使用动态分配，避免显式分配。

12. 重新均衡分片数据

如有必要，可以通过在分片间移动数据来达到负载均衡。举个例子，许多读者可能听一些大型图片分享网站或流行社区网站的开发者提到过用于分片间移动用户数据的工具。

在分片间移动数据的好处很明显。例如，当需要升级硬件时，可以将用户数据从旧分片转移到新分片上，而无须暂停整个分片的服务或将其设置为只读。

然而，我们也应该尽量避免重新均衡分片数据，因为这可能会影响用户使用。在分片间转移数据也使得为应用增加新特性更加困难，因为新特性可能还需要包含针对重新均衡脚本的升级。如果分片足够小，就无须这么做；也可以经常移动整个分片来重新均衡负载，这比移动分片中的部分数据要容易得多（并且以每行数据开销来衡量的话，更有效率）。 545

一个较好的策略是使用动态分片策略，并将新数据随机分配到分片中。当一个分片快满时，可以设置一个标志位，告诉应用不要再往这里放数据了。如果未来需要向分片中放入更多数据，可以直接把标记位清除。

假设安装了一个新的 MySQL 节点，上面有 100 个分片。先将它们的标记设置为 1，这样应用就知道它们正准备接受新数据。一旦它们的数据足够多时（例如，每个分片 10 000 个用户），就把标记位设置为 0。之后，如果节点因为大量废弃账号导致负载不足，可以重新打开一些分片向其中增加新用户。

如果升级应用并且增加的新特性会导致每个分片的查询负载升高，或者只是算错了负载，可以把一些分片移到新节点来减轻负载。缺点是操作期间整个分片会变成只读或者处于离线状态。这需要根据实际情况来看是否能接受。

另外一种使用得较多的策略是为每个分片设置两台备库，每个备库都有该分片的完整数据。然后每个备库负责其中一半的数据，并完全停止在主库上查询。这样每个备库都会有一半它不会用到的数据；我们可以使用一些工具，例如 Percona Toolkit 的 *pt-archiver*，在后台运行，移除那些不再需要的数据。这种办法很简单并且几乎不需要停机。

13．生成全局唯一 ID

当希望把一个现有系统转换为分片数据存储时，经常会需要在多台机器上生成全局唯一 ID。单一数据存储时通常可以使用 AUTO_INCREMENT 列来获取唯一 ID。但涉及多台服务器时就不凑效了。以下几种方法可以解决这个问题：

使用 `auto_increment_increment` 和 `auto_increment_offset`

这两个服务器变量可以让 MySQL 以期望的值和偏移量来增加 `AUTO_INCREMENT` 列的值。举一个最简单的场景，只有两台服务器，可以配置这两台服务器自增幅度为 2，其中一台的偏移量设置为 1，另外一台为 2（两个都不可以设置为 0）。这样一台服务器总是包含偶数，另外一台则总是包含奇数。这种设置可以配置到服务器的每一个表里。

这种方法简单，并且不依赖于某个节点，因此是生成唯一 ID 的比较普遍的方法。但

546 这需要非常仔细地配置服务器。很容易因为配置错误生成重复数字，特别是当增加服务器需要改变其角色，或进行灾难恢复时。

全局节点中创建表

在一个全局数据库节点中创建一个包含 AUTO_INCREMENT 列的表，应用可以通过这个表来生成唯一数字。

使用 *memcached*

在 *memcached* 的 API 中有一个 incr() 函数，可以自动增长一个数字并返回结果。另外也可以使用 Redis。

批量分配数字

应用可以从一个全局节点中请求一批数字，用完后再申请。

使用复合值

可以使用一个复合值来做唯一 ID，例如分片号和自增数的组合。具体参阅之前的章节。

使用 GUID 值

可以使用 UUID() 函数来生成全局唯一值。注意，尽管这个函数在基于语句的复制时不能正确复制，但可以先获得这个值，再存放到应用的内存中，然后作为数字在查询中使用。GUID 的值很大并且不连续，因此不适合做 InnoDB 表的主键。具体参考“和 InnoDB 主键一致地插入行”。在 5.1 及更新的版本中还有一个函数 UUID_SHORT()，能够生成连续的值，并使用 64 位代替了之前的 128 位。

如果使用全局分配器来产生唯一 ID，要注意避免单点争用成为应用的性能瓶颈。

虽然 *memcached* 方法执行速度快（每秒数万个值），但不具备持久性。每次重启 *memcached* 服务都需要重新初始化缓存里的值。由于需要首先找到所有分片中的最大值，因此这一过程非常缓慢并且难以实现原子性。

14. 分片工具

在设计数据分片应用时，首先要做的事情是编写能够查询多个数据源的代码。

如果没有任何抽象层，直接让应用访问多个数据源，那绝对是一个很差的设计，因为这会增加大量的编码复杂性。最好的办法是将数据源隐藏在抽象层中。这个抽象层主要完成以下任务：

- 连接到正确的分片并执行查询。
- 分布式一致性校验。
- 547 跨分片结果集聚合。
- 跨分片关联操作。

- 锁和事务管理。
- 创建新的数据分片（或者至少在运行时找到新分片）并重新平衡分片（如果有时间实现）。

你可能不需要从头开始构建分片结构。有一些工具和系统可以提供一些必要的功能或专门设计用来实现分片架构。

Hibernate Shards（*http://shards.hibernate.org*）是一个支持分片的数据库抽象层，基于 Java 语言的开源的 Hibernate ORM 库扩展，由谷歌提供。它在 Hibernate Core 接口上提供了分片感知功能，所以应用无须专门为分片设计；事实上，应用甚至无须知道它正在使用分片。Hibernate Shards 通过固定分配策略向分片分配数据。另外一个基于 Java 的分片系统是 HiveDB（*http://www.hivedb.org*）。

如果使用的是 PHP 语言，可以使用 Justin Swanhart 提供的 Shard-Query 系统（*http://code.google.com/p/shard-query/*），它可以自动分解查询，并发执行，并合并结果集。另外一些有同样用途的商用系统有 ScaleBase（*http://www.scalebase.com*）、ScalArc（*http://www.scalarc.com*），以及 dbShards（*http://www.dbshards.com*）。

Sphinx 是一个全文检索引擎，虽然不是分片数据存储和检索系统，但对于一些跨分片数据存储的查询依然有用。Sphinx 可以并行查询远程系统并聚合结果集。在附录 F 中会详细讨论 Sphinx。

11.2.5 通过多实例扩展

一个分片较多的架构可能会更有效地利用硬件。我们的研究和经验表明 MySQL 并不能完全发挥现代硬件的性能。当扩展到超过 24 个 CPU 核心时，MySQL 的性能开始趋于平缓，不再上升。当内存超过 128GB 时也同样如此，MySQL 甚至不能完全发挥诸如 Virident 或 Fusion-io 卡这样的高端 PCIe flash 设备的 I/O 性能。

不要在一台性能强悍的服务器上只运行一个服务器实例，我们还有别的选择。你可以让数据分片足够小，以使每台机器上都能放置多个分片（这也是我们一直提倡的），每台服务器上运行多个实例，然后划分服务器的硬件资源，将其分配给每个实例。

这样做尽管比较烦琐，但确实有效。这是一种向上扩展和向外扩展的组合方案。也可以用其他方法来实现——不一定需要分片——但分片对于在大型服务器上的联合扩展具有天然的适应性。

一些人倾向于通过虚拟化技术来实现合并扩展，这有它的好处。但虚拟化技术本身有很 548
大的性能损耗。具体损耗多少取决于具体的技术，但通常都比较明显，尤其是 I/O 非常

快的时候损耗会非常惊人。另一种选择是运行多个 MySQL 实例，每个实例监听不同的网络端口，或绑定到不同的 IP 地址。

我们已经在一台性能强悍的硬件上获得了 10 倍或 15 倍的合并系数。你需要平衡管理复杂度代价和更优性能的收益，以决定哪种方法是最优的。

这时候网络可能会成为瓶颈——这个问题大多数 MySQL 用户都不会遇到。可以通过使用多块网卡并进行绑定来解决这个问题。但 Linux 内核可能会不理想，这取决于内核版本，因为老的内核对每个绑定设备的网络中断只能使用一个 CPU。因此不要把太多的连线绑定到很少的虚拟设备上，否则会遇到内核层的网络瓶颈。新的内核在这一方面会有所改善，所以需要检查你的系统版本，以确定该怎么做。

另一个方法是将每个 MySQL 实例绑定到特定的 CPU 核心上。这有两点好处：第一，由于 MySQL 内部的可扩展性限制，当核心数较少时，能够在每个核心上获得更好的性能；第二，当实例在多个核心上运行线程时，由于需要在多核心上同步共享数据，因而会有一些额外的开销。这可以避免硬件本身的可扩展性限制。限制 MySQL 到少数几个核心能够帮助减少 CPU 核心之间的交互。注意到反复出现的问题了没？将进程绑定到具有相同物理套接字的核心上可以获得最优的效果。

11.2.6 通过集群扩展

理想的扩展方案是单一逻辑数据库能够存储尽可能多的数据，处理尽可能多的查询，并如期望的那样增长。许多人的第一想法就是建立一个"集群"或者"网格"来无缝处理这些事情，这样应用就无须去做太多工作，也不需要知道数据到底存在哪台服务器上。随着云计算的流行，自动扩展——根据负载或数据大小变化动态地在集群中增加 / 移除服务器——变得越来越有趣。

在本书第二版时，我们遗憾地看到已有的技术无法完成这一任务。从那时开始，出现了许多被称为 NoSQL 的技术。许多 NoSQL 的支持者发表了一些奇怪且未经证实的观点，例如"关系模型无法进行扩展"，或者"SQL 无法扩展"。随着新概念的出现，也出现了一些新的术语。最近谁没有听说过最终一致性、BASE、矢量时钟，或者 CAP 理论呢？

但随着时间推移，理性开始逐渐回归。经验表明许多 NoSQL 数据库太过于简单，并
549 且无法完成很多工作[注9]。同时一些基于 SQL 的技术开始出现——例如 451 集团（451 Group）的 Matt Aslett 所提到的 NewSQL 数据库。SQL 和 NewSQL 到底有什么区别呢？NewSQL 数据库中 SQL 及相关技术都不应该成为问题。而可扩展性问题在关系型数据库中是一个实现上的难题，但新的实现正表现出越来越好的结果。

注 9： Yeah, yeah, 我们知道，为你的工作选择正确的工具。这里引用显而易见但听起来很有意义的评论。

所有的旧事物都变成新的了吗？是，但也不是。许多关系型数据库集群的高性能设计正在被构建到系统的更低层，在NoSQL数据库中，特别是使用键—值存储时，这一点很明显。例如 NDB Cluster 并不是一个 SQL 数据库；它是一个可扩展的数据库，使用其原生 API 来控制，通常是使用 NoSQL，但也可以通过在前端使用 MySQL 存储引擎来支持 SQL。它是一个完全分布式、非共享高性能、自动分片并且不存在单点故障的事务型数据库服务器。最近几年正变得更强大、更复杂，用途也更广泛。同时，NoSQL 数据库也逐渐看起来越来越像关系型数据库。有些甚至还开发了类 SQL 查询语言。未来典型的集群数据库可能更像是 SQL 和 NoSQL 的混合体，有多种存取机制来满足不同的使用需求。所以，我们在从 NoSQL 中汲取优点，但 SQL 仍然会保留在集群数据库中。

在写作本书时，和 MySQL 结合在一起的集群或分布式数据库技术大致包括：NDB Cluster、Clustrix、Percona XtraDB Cluster、Galera、Schooner Active Cluster、Continuent Tungsten、ScaleBase、ScaleArc、dbShards、Xeround、Akiban、VoltDB，以及 GenieDB。这些或多或少以 MySQL 为基础，或通过 MySQL 进行控制，或是和 MySQL 相关。本书会讲到这其中的一部分——例如，在第 13 章我们会讲到 Xeround，在第 10 章我们讲到了 Continuent Tungsten 和其他几种技术——这里我们同样会对其中的几个进行描述。

在开始前，需要指出，可扩展性、高可用性、事务性等是数据库系统的不同特性。许多人会感到困惑并将这些当作是相同的东西，但事实上不是。本章我们主要集中讨论可扩展性。但事实上，可扩展的数据库并不一定非常优秀，除非它能保证高性能，谁愿意牺牲高可用性来进行扩展呢？这些特性的组合堪称数据库的必杀技，但这很难实现。当然这不是本章要讨论的内容。

最后，除 NDB Cluster 外，大多数 NewSQL 集群产品都是比较新的事物。我们还没有看到足够多的生产环境部署以完全获知其优点和限制。尽管它们提到了 MySQL 协议或其 550
他与 MySQL 相关的地方，但它们毕竟不是 MySQL，因此不在本书讨论的范围内。我们仅仅稍微提一下，由你自己来判断它们是否适用。

1. MySQL Cluster（NDB Cluster）

MySQL Cluster 是两项技术的结合：NDB 数据库，以及作为 SQL 前端的 MySQL 存储引擎。NDB 是一个分布式、具备容错性、非共享的数据库，提供同步复制以及节点间的数据自动分片。NDB Cluset 存储引擎将 SQL 转换为 NDB API 调用，但遇到 NDB 不支持的操作时，就会在 MySQL 服务器上执行（NDB 是一个键—值数据存储，无法执行类似联接或聚合的复杂操作）。

NDB 是一个非常复杂的数据库，和 MySQL 几乎完全不同。在使用 NDB 时甚至可以不

需要 MySQL：你可以把它作为一个独立的键—值数据库服务器。它的亮点包括非常高的写入和按键查询吞吐量。NDB 可以基于键的哈希自动决定哪个节点应该存储给定的数据。当通过 MySQL 来控制 NDB 时，行的主键就是键，其他的列是值。

因为它基于一些新的技术，并且集群具有容错性和分布式特性，所以管理 NDB 需要非常专业和特殊的技能。有许多动态变化的部分，还有类似升级集群或增加节点的操作必须正确执行以防止意外的问题。NDB 是一项开源技术，但也可以从 Oracle 购买商业支持。商业支持中包括能够获得专门的集群管理产品 Cluster Manager，可以自动执行一些枯燥且棘手的任务。（Severalnines 同样提供了一个集群管理产品，参见 *http://www.severalnines.com*）。

MySQL Cluster 正在迅速地增加越来越多的特性和功能。例如在最近的版本中，它开始支持更多类型的集群变更而无须停机操作，并且能够在数据存储的节点上执行一些特定类型的查询，以减少数据传递给 MySQL 层并在其中执行查询的必要性。（这个特性已由关联下推（push-down join）更名为自适应查询本地化（adaptive query localization）。）

NDB 曾经相对其他 MySQL 存储引擎具有完全不同的性能特性，但最近的版本更加通用化了。它正在成为越来越多应用的更好的解决方案，包括游戏和移动应用。我们必须强调，NDB 是一项重要的技术，能够支持全球最大的关键应用，这些应用处于极高的负载下，具有非常严苛的延迟要求以及不间断要求。举个例子，世界上任何一个通过移动电话网络呼叫的电话使用的就是 NDB，并且不是临时方案——对于许多移动电话提供商而言，它是一个主要的并且非常重要的数据库。

NDB 需要一个快速且可靠的网络来连接节点。为了获得最好的性能，最好使用特定的高速连接设备。由于大多数情况下需要内存操作，因此服务器间需要大量的内存。

551 那么它有什么缺点呢？复杂查询现在支持得还不是很好，例如那些有很多关联和聚合的查询。所以不要指望用它来做数据仓库。NDB 是一个事务型系统，但不支持 MVCC，所以读操作也需要加锁，也不做任何的死锁检测。如果发生死锁，NDB 就以超时返回的方式来解决。还有很多你应该知道的要点和警告，可以专门写一本书了。（有一些关于 MySQL Cluster 的书，但大多数都过时了，最好的办法是阅读手册。）

2．Clustrix

Clustrix（*http://www.clustrix.com*）是一个分布式数据库，支持 MySQL 协议，所以它可以直接替代 MySQL。除了协议外，它是一个全新的技术，并非建立在 MySQL 的基础之上。它是一个完全支持 ACID，支持 MVCC 的事务型 SQL 数据库，主要用于 OLTP 负载场景。Clustrix 在节点间进行数据分片以满足容错性，并对查询进行分发，在节点上并发执行，

而不是将所有节点上取得的数据集中起来执行。集群可以在线扩展节点来处理更多的数据或负载。在某些方面 Clustrix 和 MySQL Cluster 很像；关键的不同点是，Clustrix 是完全分布式执行并且缺少顶层的“代理”或者集群前端的查询协调器（query coordinator）。Clustrix 本身能够理解 MySQL 协议，所以无须 MySQL 来进行协议转换。相比较而言，MySQL cluster 是由三个部分组成的：MySQL，NDB 集群存储引擎，以及 NDB。

我们的实验评估和性能测试表明，Clustrix 能够提供高性能和可扩展性。Clustrix 看起来是一项比较有前景的技术，我们将继续观察和评估。

3．ScaleBase

ScaleBase（*http://www.scalebase.com*）是一个软件代理，处于应用和多个后端 MySQL 服务器之间。它会把发起的查询进行分裂，并将其分发到后端服务器并发执行，然后汇集结果返回给应用。不过在写作本书时，我们还没有使用该产品的经验。另外的竞争产品有 ScaleArc（*http://www.calearc.com*）和 dbShards（*http://www.dbshards.com*）。

4．GenieDB

GenieDB（*http://www.geniedb.com*）最开始用于地理上分布部署的 NoSQL 文档存储。现在它也有一个 SQL 层，可以通过 MySQL 存储引擎进行控制。它包含了很多技术，包括本地内存缓存、消息层，以及持久化磁盘数据存储。将这些技术汇集在一起，就可以使用松散的最终一致性，让应用在本地快速执行查询，或是通过分布式集群（会增加网络延迟）来保证最新的数据视图。

通过存储引擎实现的 MySQL 兼容层不能提供 100% 的 MySQL 特性，但对于支持类似 552
Joomla!、WordPress，以及 Drupal 这样的应用已经够用了。MySQL 存储引擎的用处主要是使 GenieDB 能够结合存储引擎获得对 ACID 的支持，例如 InnoDB。GenieDB 本身并不是 ACID 数据库。

我们还没用应用过 GenieDB，也没有看到任何生产环境部署。

5．Akiban

对 Akiban（*http://www.akiban.com*）最好的描述应该是查询加速器。它通过存储物理数据来匹配查询模式，使得低开销的跨表关联操作成为可能。尽管类似反范式化（denormalization），但数据层并不是冗余的，所以这和预先计算关联并存储结果的方式是不同的。关联表中元组是互相交错的，所以能够按照关联顺序进行顺序扫描。这就要求管理员确定查询模式能够从所谓的“表组”（table grouping）技术中受益，并需要为查询优化设计表组。目前建议的系统架构是将 Akiban 配置为 MySQL 主库的备库，并用

它来为可能较慢的查询提供服务。加速系数是一到两个数量级。但是我们还没有看到生产环境部署或者相关的实验评估。[注10]

11.2.7 向内扩展

处理不断增长的数据和负载最简单的办法是对不再需要的数据进行归档和清理。这种操作可能会带来显著的成效，具体取决于工作负载和数据特性。这种做法并不用来代替其他策略，但可以作为争取时间的短期策略，也可以作为处理大数据量的长期计划之一。

在设计归档和清理策略时需要考虑到如下几点。

对应用的影响

一个设计良好的归档系统能够在不影响事务处理的情况下，从一个高负载的 OLTP 服务器上移除数据。这里的关键是能高效地找到要删除的行，然后一小块一小块地移除。通常需要平衡一次归档的行数和事务的大小，以找到一个锁竞争和事务负载量的平衡。还需要设计归档任务在必要的时候让步于事务处理。

553 *要归档的行*

当知道某些数据不再使用后，就可以立刻清理或归档它们。也可以设计应用去归档那些几乎不怎么使用的数据。可以把归档的数据置于核心表附近，通过视图来访问，或完全转移到别的服务器上。

维护数据一致性

当数据间存在联系时，会导致归档和清理工作更加复杂。一个设计良好的归档任务能够保证数据的逻辑一致性，或至少在应用需要时能够保证一致，而无须在大量事务中包含多个表。

当表之间存在关系时，哪个表首先归档是个问题。在归档时需要考虑孤立行的影响。可以选择违背外键约束（可以通过执行 `SET FOREIGN_KEY_CHECKS=0` 禁止 InnoDB 的外键约束）或暂时把“悬空指针”（dangling pointer）记录放到一边。如果应用层认为这些相关联的表具有层次关系，那么归档的顺序也应该和它一样。例如，如果应用总是先检查订单再检查发货单，就先归档订单。应用应该看不到孤立的发货单，因此接下来就可以将发货单归档。

避免数据丢失

如果是在服务器间归档，归档期间可能就无法做分布式事务处理，也有可能将数据归档到 MyISAM 或其他非事务型的存储引擎中。因此，为了避免数据丢失，在从源表中删除时，要保证已经在目标机器上保存。将归档数据单独写到一个文件里也是个好主意。可以将归档任务设计为能够随时关闭或重启，并且不会引起不一致或索

注 10：我们将 Akiban 包含在集群数据库列表中可能并不准确，因为它并不是真正的集群数据库。但在某种程度上它和其他一些 NewSQL 数据库很像。

引冲突之类的错误。

解除归档（unarchiving）

可以通过一些解除归档策略来减少归档的数据量。它可以帮助你归档那些不确定是否需要的数据，并在以后可以通过选项进行回退。如果可以设置一些检查点让系统来检查是否有需要归档的数据，那么这应该是一个很容易实现的策略。例如，要对不活跃的用户进行归档，检查点就可以设置在登录验证时。如果因为用户不存在导致登录失败，可以去检查归档数据中是否存在该用户，如果有，则从中取出来并完成登录。

> Percona Toolkit 包含的工具 *pt-archiver* 能够帮助你有效地归档和清理 MySQL 表，但不提供解除归档功能。

保持活跃数据独立 554

即使并不真的把老数据转移到别的服务器，许多应用也能受益于活跃数据和非活跃数据的隔离。这有助于高效利用缓存，并为活跃和不活跃的数据使用不同的硬件或应用架构。下面列举了几种做法：

将表划分为几个部分

分表是一种比较明智的办法，特别是整张表无法完全加载到内存时。例如，可以把 `users` 表划分为 `active_users` 和 `inactive_users` 表。你可能认为这并不需要，因为数据库本身只缓存"热"数据，但事实上这取决于存储引擎。如果用的是 InnoDB，每次缓存一页，而一页能存储 100 个用户，但只有 10% 是活跃的，那么这时候 InnoDB 可能认为所有的页都是"热"的——因此每个"热"页的 90% 将被浪费掉。将其拆成两个表可以明显改善内存利用率。

MySQL 分区

MySQL 5.1 本身提供了对表进行分区的功能，能够帮助把最近的数据留在内存中。第 7 章详细介绍了分区表。

基于时间的数据分区

如果应用不断有新数据进来，一般新数据总是比旧数据更加活跃。例如，我们知道博客服务的流量大多是最近七天发表的文章和评论。更新的大部分是相同的数据集。因此这些数据被完整地保留在内存中，使用复制来保证在主库失效时有一份可用的备份。其他数据则完全可以放到别的地方去。

我们也看到过这样一种设计，在两个节点的分片上存储用户数据。新数据总是进入"活跃"节点，该节点使用更大的内存和快速硬盘，另外一个节点存储旧数据，使用

非常大（但比较慢）的硬盘。应用假设不太会需要旧数据。对于很多应用而言这是合理的假设，依靠 10% 的最新数据能够满足 90% 或更多的请求。

可以通过动态分片来轻松实现这种策略。例如，分片目录表可能定义如下：

```
CREATE TABLE users (
   user_id           int unsigned not null,
   shard_new         int unsigned not null,
   shard_archive     int unsigned not null,
   archive_timestamp timestamp,
   PRIMARY KEY (user_id)
);
```

通过一个归档脚本将旧数据从活跃节点转移到归档节点，当移动用户数据到归档节点时，更新 archive_timestamp 列的值。shard_new 和 shard_archive 列记录存储数据的分片号。

555 11.3 负载均衡

负载均衡的基本思路很简单：在一个服务器集群中尽可能地平均负载量。通常的做法是在服务器前端设置一个负载均衡器（一般是专门的硬件设备）。然后负载均衡器将请求的连接路由到最空闲的可用服务器。图 11-9 显示了一个典型的大型网站负载均衡设置，其中一个负载均衡器用于 HTTP 流量，另一个用于 MySQL 访问。

负载均衡有五个常见目的。

可扩展性

负载均衡对某些扩展策略有所帮助，例如读写分离时从备库读数据。

556 **高效性**

负载均衡有助于更有效地使用资源，因为它能够控制请求被路由到何处。如果服务器处理能力各不相同，这就尤为重要：你可以把更多的工作分配给性能更好的机器。

可用性

一个灵活的负载均衡解决方案能够使用时刻保持可用的服务器。

透明性

客户端无须知道是否存在负载均衡设置，也不需要关心在负载均衡器的背后有多少机器，它们的名字是什么。负载均衡器给客户端看到的只是一个虚拟的服务器。

一致性

如果应用是有状态的（数据库事务，网站会话等），那么负载均衡器就应将相关的查询指向同一个服务器，以防止状态丢失。应用无须去跟踪到底连接的是哪个服务器。

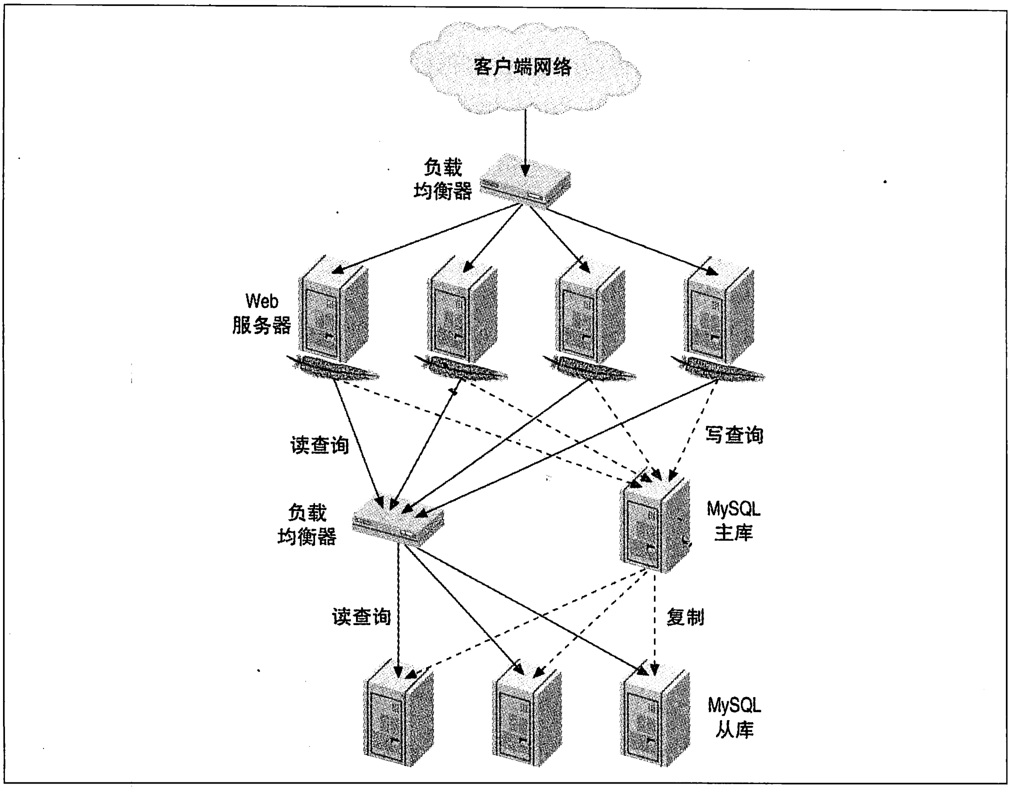


图11-9：一个典型的读密集型网站负载均衡架构

在与 MySQL 相关的领域里，负载均衡架构通常和数据分片及复制紧密相关。你可以把负载均衡和高可用性结合在一起，部署到应用的任一层次上。例如，可以在 MySQL Cluster 集群的多个 SQL 节点上做负载均衡，也可以在多个数据中心间做负载均衡，其中每个数据中心又可以使用数据分片架构，每个节点实际上是拥有多个备库的主—主复制对结构，这里又可以做负载均衡。对于高可用性策略也同样如此：在一个架构里可以配置多层的故障转移机制。

负载均衡有许多微妙之处，举个例子，其中一个挑战就是管理读 / 写策略。有些负载均衡技术本身能够实现这一点，但其他的则需要应用自己知道哪些节点是可读的或可写的。

在决定如何实现负载均衡时，应该考虑到这些因素。有许多负载均衡解决方案可以使用，从诸如 Wackamole（*http://www.backhand.org/wackamole/*）这样基于端点的（peer-based）实现，到 DNS、LVS（Linux Virtual Server，*http://www.linuxvirtualserver.org*）、硬件负载均衡器、TCP 代理、MySQL Proxy，以及在应用中管理负载均衡。

在我们的客户中，最普遍的策略是使用硬件负载均衡器，大多是使用 HAProxy（*http://haproxy.1wt.eu*），它看起来很流行并且工作得很好。还有一些人使用 TCP 代理，例如 Pen（*http://siag.nu/pen/*）。但 MySQL Proxy 用得并不多。

11.3.1 直接连接

有些人认为负载均衡就是配置在应用和 MySQL 服务器之间的东西。但这并不是唯一的
557
负载均衡方法。你可以在保持应用和 MySQL 连接的情况下使用负载均衡。事实上，集中化的负载均衡系统只有在存在一个对等置换的服务器池时才能很好工作。如果应用需要做一些决策，例如在备库上执行读操作是否安全，就需要直接连接到服务器。

除了可能出现的一些特定逻辑，应用为负载均衡做决策是非常高效的。例如，如果有两个完全相同的备库，你可以使用其中的一个来处理特定分片的数据查询，另一个处理其他的查询。这样能够有效利用备库的内存，因为每个备库只会缓存一部分数据。如果其中一个备库失效，另外一个备库拥有所有的数据，仍然能提供服务。

接下来的小节将讨论一些应用直连的常见方法，以及在评估每一个选项时的注意点。

1. 复制上的读 / 写分离

MySQL 复制产生了多个数据副本，你可以选择在备库还是主库上执行查询。由于备库复制是异步的，因此主要的难点是如何处理备库上的脏数据。应该将备库用作只读的，而主库可以同时处理读和写查询。

通常需要修改应用以适应这种分离需求。然后应用就可以使用主库来进行写操作，并将读操作分配到主库和备库上；如果不太关心数据是否是脏的，可以使用备库，而对需要即时数据的请求使用主库。我们将这称为读 / 写分离。

如果使用的是主动—被动模式的主—主复制对，同样也要考虑这个问题。使用这种配置时，只有主动服务器接受写操作。如果能够接受读到脏数据，可以将读分配给被动服务器。

最大的问题是如何避免由于读了脏数据引起的奇怪问题。一个典型的例子是当一个用户做了某些修改，例如增加了一条博客文章的评论，然后重新加载页面，但并没有看到更新，因为应用从备库读取到了脏的数据。

比较常见的读 / 写分离方法如下：

基于查询分离

　最简单的分离方法是将所有不能容忍脏数据的读和写查询分配到主动或主库服务器上。其他的读查询分配到备库或被动服务器上。该策略很容易实现，但事实上无法

有效地使用备库，因为只有很少的查询能容忍脏数据。

基于脏数据分离

这是对基于查询分离方法的小改进。需要做一些额外的工作，让应用检查复制延迟，以确定备库数据是否太旧。许多报表类应用都使用这个策略：只需要晚上加载的数据复制到备库即可，它们并不关心是不是100%跟上了主库。 558

基于会话分离

另一个决定能否从备库读数据的稍微复杂一点的方法是判读用户自己是否修改了数据。用户不需要看到其他用户的最新数据，但需要看到自己的更新。可以在会话层设置一个标记位，表明做了更新，就将该用户的查询在一段时间内总是指向主库。这是我们通常推荐的策略，因为它是在简单和有效性之间的一种很好的妥协。

如果有足够的想象力，可以把基于会话的分离方法和复制延迟监控结合起来。如果用户在10秒前更新了数据，而所有备库延迟在5秒内，就可以安全地从备库中读取数据。但为整个会话选择同一个备库是一个很好的主意，否则用户可能会奇怪有些备库的更新速度比其他服务器要慢。

基于版本分离

这和基于会话的分离方法相似：你可以跟踪对象的版本号以及/或者时间戳，通过从备库读取对象的版本或时间戳来判断数据是否足够新。如果备库的数据太旧，可以从主库获取最新的数据。即使对象本身没有变化，但如果是顶层对象，只要下面的任何对象有变化，也可以增加版本号，这简化了脏数据检查（只需要检查顶层对象一处就能判断是否有更新）。例如，在用户发表了一篇新文章后，可以更新用户的版本。这样就会从主库去读取数据了。

基于全局版本/会话分离

这个办法是基于版本分离和基于会话分离的变种。当应用执行写操作时，在提交事务后，执行一次 `SHOW MASTER STATUS` 操作。然后在缓存中存储主库日志坐标，作为被修改对象以及/或者会话的版本号。当应用连接到备库时，执行 `SHOW SLAVE STATUS` 并将备库上的坐标和缓存中的版本号相对比。如果备库相比记录点更新，就可以安全地读取备库数据。

大多数读/写分离解决方案都需要监控复制延迟来决策读查询的分配，不管是通过复制或负载均衡器，或是一个中间系统。如果这么做，需要注意通过 `SHOW SLAVE STATUS` 得到的 `Seconds_behind_master` 列的值并不能准确地用于监控延迟。（详情参阅第10章）。Percona Toolkit 中的 *pt-heartbeat* 工具能够帮助监控延迟，并维护元数据，例如二进制日志位置，这可以减轻之前我们讨论的一些策略存在的问题。 559

如果不在乎用昂贵的硬件来承载压力，也就可以不使用复制来扩展读操作，这样当然更简单。这可以避免在主备上分离读的复杂性。有些人认为这很有意义；也有人认为会浪

费硬件。这种分歧是由于不同的目的引起的：你是只需要可扩展性，还是要同时具有可扩展性和高利用率？如果需要高利用率，那么备库除了保存数据副本外还需要承担其他任务，就不得不处理这些额外的复杂度。

2．修改应用的配置

还有一个分发负载的方法是重新配置应用。例如，你可以配置多个机器来分担生成大报表操作的负载。每台机器可以配置成连接到不同的 MySQL 备库，并为第 *N* 个用户或网站生成报表。

这样的系统很容易实现，但如果需要修改一些代码——包括配置文件修改——会变得脆弱且难以处理。硬编码有着固有的限制，需要在每台服务器上修改硬编码，或者在一个中心服务器上修改，然后通过文件副本或代码控制更新命令“发布”到其他服务器上。如果将配置存储在服务器或缓存中，就可以避免这些麻烦。

3．修改 DNS 名

这是一个比较粗糙的负载均衡技术，但对于一些简单的应用，为不同的目的创建 DNS 还是很实用的。你可以为不同的服务器指定一个合适的名字。最简单的方法是只读服务器拥有一个 DNS 名，而给负责写操作的服务器起另外一个 DNS 名。如果备库能够跟上主库，那就把只读 DNS 名指定给备库，当出现延迟时，再将该 DNS 名指定给主库。

这种 DNS 技术非常容易实现，但也有很多缺点。最大的问题是无法完全控制 DNS。

- 修改 DNS 并不是立刻生效的，也不是原子的。将 DNS 的变化传递到整个网络或在网络间传播都需要比较长的时间。
- DNS 数据会在各个地方缓存下来，它的过期时间是建议性质的，而非强制的。
- 可能需要应用或服务器重启才能使修改后的 DNS 完全生效。
- 多个 IP 地址共用一个 DNS 名并依赖于轮询行为来均衡请求，这并不是一个好主意。因为轮询行为并不总是可预知的。
- DBA 可能没有权限直接访问 DNS。

560 除非应用非常简单，否则依赖于不受控制的系统会非常危险。你可以通过修改 */etc/hosts* 文件而非 DNS 来改善对系统的控制。当发布一个对该文件的更新时，会知道该变更已经生效。这比等待缓存的 DNS 失效要好得多。但这仍然不是理想的办法。

我们通常建议人们构建一个完全不依赖 DNS 的应用。即使应用很简单也适用，因为你无法预知应用会增长到多大规模。

4. 转移 IP 地址

一些负载均衡解决方案依赖于在服务器间转移虚拟地址[注11]，一般能够很好地工作。这听起来和修改 DNS 很像，但完全是两码事。服务器不会根据 DNS 名去监听网络流量，而是根据指定的 IP 地址去监听流量，所以转移 IP 地址允许 DNS 名保持不变。你可以通过 ARP（地址解析协议）命令强制使 IP 地址的更改快速而且原子性地通知到网络上。

我们看过的使用最普遍的技术是 Pacemaker，这是 Linux-HA 项目的 Heartbeat 工具的继承者。你可以使用单个 IP 地址，为其分配一个角色，例如 read-only，当需要在机器间转移 IP 地址时，它能够感知到。其他类似的工具包括 LVS 和 Wackamole。

一个比较方便的技术是为每个物理服务器分配一个固定的 IP 地址。该 IP 地址固定在服务器上，不再改变。然后可以为每个逻辑上的“服务”使用一个虚拟 IP 地址。它们能够很方便地在服务器间转移，这使得转移服务和应用实例无须再重新配置应用，因此更加容易。即使不怎么经常转移 IP 地址，这也是一个很好的特性。

11.3.2 引入中间件

迄今为止，我们所讨论的方案都假定应用跟 MySQL 服务器是直接相连的。但是许多负载均衡解决方案都会引入一个中间件，作为网络通信的代理。它一边接受所有的通信请求，另一边将这些请求派发到指定的服务器上，然后把执行结果发送回请求的机器上。中间件可以是硬件设备或是软件[注12]。图 11-10 描述了这种架构。这种解决方案通常能工作得很好，当然除非为负载均衡器本身增加冗余，这样才能避免单点故障引起的整个系统瘫痪。从开源软件，如 HAProxy，到许多广为人知的商业系统，有许多负载均衡器得到了成功的应用。 561

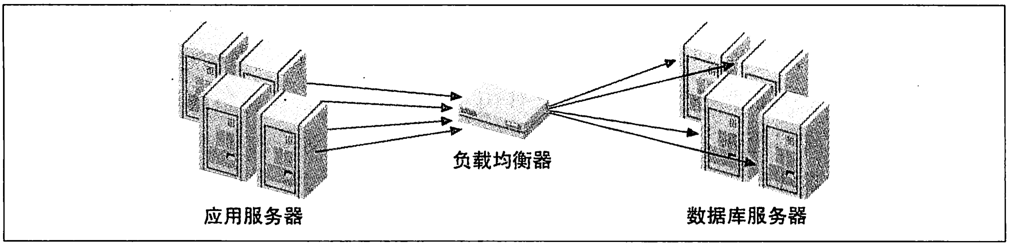


图11-10：作为中间件的负载均衡器

注 11： 虚拟 IP 地址不是直接连接到任何特定的计算机或网络端口，而是“漂浮”在计算机之间。

注 12： 你可以把诸如 LVS 这样的解决方案配置成只有应用需要创建一个新连接时才参与进来，此后不再作为中间件。

1. 负载均衡器

在市场上有许多负载均衡硬件和软件，但很少有专门为 MySQL 服务器设计的[注13]。Web 服务器通常更需要负载均衡，因此许多多用途的负载均衡设备都会支持 HTTP，而对其他用途则只有一些很少的基本特性。MySQL 连接都只是正常的 TCP/IP 连接，所以可以在 MySQL 上使用多用途负载均衡器。但由于缺少 MySQL 专有的特性，因此会多一些限制。

- 除非负载均衡器知道 MySQL 的真实负载，否则在分发请求时可能无法做到很好的负载均衡。不是所有的请求都是等同的，但多用途负载均衡器通常对所有的请求一视同仁。
- 许多负载均衡器知道如何检查一个 HTTP 请求并把会话“固定”到一个服务器上以保护在 Web 服务器上的会话状态。MySQL 连接也是有状态的，但负载均衡器可能并不知道如何把所有从单个 HTTP 会话发送的连接请求“固定”到一个 MySQL 服务器上。这会损失一部分效率。（如果单个会话的请求都是发到同一个 MySQL 服务器，服务器的缓存会更有效率。）
- 连接池和长连接可能会阻碍负载均衡器分发连接请求。例如，假如一个连接池打开了预先配置好的连接数，负载均衡器在已有的四个 MySQL 服务器上分发这些连接。现在增加了两个以上的 MySQL 服务器。由于连接池不会请求新连接，因而新的服务器会一直空闲着。池中的连接会在服务器间不公平地分配负载，导致一些服务器
562 超出负载，一些则几乎没有负载。可以在多个层面为连接设置失效时间来缓解这个问题，但这很复杂并且很难做到。连接池方案只有它们本身能够处理负载均衡时才能工作得很好。
- 许多多用途负载均衡器只会针对 HTTP 服务器做健康和负载检查。一个简单的负载均衡器最少能够核实服务器在一个 TCP 端口上接受的连接数。更好的负载均衡器能够自动发起一个 HTTP 请求，并检查返回值以确定这个 Web 服务器是否正常运转。MySQL 并不接受到 3306 端口的 HTTP 请求，因此需要自己来构建健康检查方法。你可以在 MySQL 服务器上安装一个 HTTP 服务器软件，并将负载均衡器指向一个脚本，这个脚本检查 MySQL 服务器的状态并返回一个对应的状态值[注14]。最重要的是检查操作系统负载（通过查看 */proc/loadavg*）、复制状态，以及 MySQL 的连接数。

2. 负载均衡算法

有许多算法用来决定哪个服务器接受下一个连接。每个厂商都有各自不同的算法，下面

注 13： MySQL Proxy 是个例外，但目前还未能证明能够很好地工作，因为它会带来一些问题，例如延迟增加以及可扩展性瓶颈。

注 14： 实际上，如果能编码实现一个监听 80 端口的程序，或者配置 *xinetd* 来调用程序，甚至不需要再安装一个 Web 服务器。

这个清单列出了一些可用的方法：

随机
: 负载均衡器随机地从可用的服务器池中选择一个服务器来处理请求。

轮询
: 负载均衡器以循环顺序发送请求到服务器，例如：A，B，C，A，B，C。

最少连接数
: 下一个连接请求分配给拥有最少活跃连接的服务器。

最快响应
: 能够最快处理请求的服务器接受下一个连接。当服务器池里同时存在快速和慢速服务器时，这很有效。即使同样的查询在不同的场景下运行也会有不同的表现，例如当查询结果已经缓存在查询缓存中，或者服务器缓存中已经包含了所需要的数据时。

哈希
: 负载均衡器通过连接的源 IP 地址进行哈希，将其映射到池中的同一个服务器上。每次从同一个 IP 地址发起请求，负载均衡器都会将请求发送给同样的服务器。只有当池中服务器数目改变时这种绑定才会发生变化。

权重
: 负载均衡器能够结合使用上述几种算法。例如，你可能拥有单 CPU 和双 CPU 的机器。双 CPU 机器有接近两倍的性能，所以可以让负载均衡器分派两倍的请求给双 CPU 机器。 563

哪种算法最优取决于具体的工作负载。例如最少连接算法，如果有新机器加入，可能会有大量连接涌入该服务器，而这时候它的缓存还没有包含热数据。本书第一版的作者曾经亲身体验了这种情况。

你需要通过测试来为你的工作负载找到最好的性能。除了正常的日常运转，还需要考虑极端情况。在比较极端的情况下——例如负载升高，修改模式，或者多台服务器下线——至少要避免系统出现重大错误。

我们这里只描述了即时处理请求的算法，无须对连接请求排队。但有时候使用排队算法可能更有效。例如，一个算法可能只维护给定的数据库服务器并发数目，同一时刻只允许不超过 N 个活跃事务。如果有太多的活跃事务，就将新的请求放到一个队列里，然后让可用服务器列表的第一个来处理它。有些连接池也支持队列算法。

3. 在服务器池中增加 / 移除服务器

增加一个服务器到池中并不是简单地插入进去，然后通知负载均衡器就可以了。你可能以为只要不是一下子涌进大量连接请求就可以了，但并不一定如此。有时候你会缓慢增

加一台服务器的负载，但一些缓存还是"冷"的服务器可能会慢到在一段时间内都无法处理任何的用户请求。如果用户浏览一个页面需要 30 秒才能返回数据，即使流量很小，这个服务器也是不可用的。有一个方法可以避免这个问题，在通知负载均衡器有新服务器加入前，可以暂时把 SELECT 查询映射到一台活跃服务器上。然后在新开启的服务器上读取和重放活跃服务器上的日志文件，或者捕捉生产服务器上的网络通信，并重放它的一部分查询。Percona Toolkit 中的 *pt-query-digest* 工具能够有所帮助。另一个有效的办法是使用 Percona Server 或 MySQL 5.6 的快速预热特性。

在配置连接池中的服务器时，要保证有足够多未使用的容量，以备在撤下服务器做维护时使用，或者当服务器失效时可以派上用场。每台服务器上都应该保留高于"足够"的容量。

要确保配置的限制值足够高，即使从池中撤出一些服务器也能够工作。举个例子，如
564 果你发现每个 MySQL 服务器一般有 100 个连接，应该设置池中每个服务器的 `max_connections` 值为 200。这样就算一半的服务器失效，服务器池整体也能处理同样数量的请求。

11.3.3 一主多备间的负载均衡

最常见的复制拓扑结构就是一个主库加多个备库。我们很难绕开这个架构。许多应用都假设只有一个目标机器用于所有的写操作，或者所有的数据都可以从单个服务器上获得。尽管这个架构不太具有很好的可扩展性，但可以通过一些办法结合负载均衡来获得很好的效果。本小节将讲述其中的一些技术。

功能分区

正如之前讨论的，对于特定的目的可以通过配置备库或一组备库来极大地扩展容量。一些比较常见的功能包括报表、分析、数据仓库，以及全文检索。在第 10 章有更多的细节。

过滤和数据分区

可以使用复制过滤技术在相似的备库上对数据进行分区（参考第 10 章）。只要数据在主库上已经被隔离到不同的数据库或表中，这种方法就可以奏效。不幸的是，没有内建的办法在行级别上进行复制过滤。你需要使用一些独创性的技术来实现这一点，例如使用触发器和一组不同的表。

即使不把数据分区到各个备库上，也可以通过对读进行分区而不是随机分配来提高缓存效率。例如，可以把对以字母 A—M 开头的用户名的读操作分配给一个给定的备库，把以 N—Z 开头的分配给另外一个。这能够更好地利用每台机器的缓存，因为分离读更可能在缓存中找到相关的数据。最好的情况下，当没有写操作时，这样

使用的缓存相当于两台服务器缓存的总和。相比之下，如果随机地在备库上分配读操作，每个机器的缓存本质上还是重复的数据，而总的有效缓存效率和一个备库缓存一样，不管你有多少台备库。

将部分写操作转移到备库

主库并不总是需要处理写操作中的所有工作。你可以分解写查询，并在备库上执行其中的一部分，从而显著减少主库的工作量。更多内容参见第 10 章。

保证备库跟上主库

如果要在备库执行某种操作，它需要即时知道数据处于哪个时间点——哪怕需要等待一会儿才能到达这个点——可以使用函数 MASTER_POS_WAIT() 阻塞直到备库赶上了设置的主库同步点。另一种替代方案是使用复制心跳来检查延迟情况；更多内容参见第 10 章。 565

同步写操作

也可以使用 MASTER_POS_WAIT() 函数来确保写操作已经被同步到一个或多个备库上。如果应用需要模拟同步复制来保证数据安全性，就可以在多个备库上轮流执行 MASTER_POS_WAIT() 函数。这就类似创建了一个“同步屏障”，当任意一个备库出现复制延迟时，都可能花费很长时间完成，所以最好在确实需要的时候才使用这种方法。（如果你的目的只是确保某些备库拥有事件，可以只等待一台备库接收到事件。MySQL 5.5 增加了半同步复制，能够支持这项技术。）

11.4 总结

正确地扩展 MySQL 并没有看起来那么美好。从第一天就建立下一个 Facebook 架构，这并不是正确的方式。最好的策略是实现应用所明确需要的，并为可能的快速增长做好预先规划，成功的规划是可以为任何必要的措施筹集资金以满足需求。

为可扩展性制定一个数学意义上的定义是很有意义的，就像为性能制定了一个精确概念一样。USL 能够提供一个有帮助的框架。如果知道系统无法做到线性扩展是因为诸如序列化或交互操作的开销，将可以帮助你避免将这些问题带入到应用中。同时，许多可扩展性问题并不是可以从数学上定义的；可能是由于组织内部的问题，例如缺少团队协作或其他不适当的问题。Neil J. Gunther 博士所写的 *Guerrilla Capacity Planning* 以及 Eliyahu M. Goldratt 写的 *The Goal* 可以帮助有兴趣的读者了解为什么系统无法扩展。

在 MySQL 扩展策略方面，典型的应用在增长到非常庞大时，通常先从单个服务器转移到向外扩展的拥有读备库的架构，再到数据分片和 / 或者按功能分区。我们并不同意那些提倡为每个应用“尽早分片，尽量分片”（shard early, shard often）的建议。这很复杂且代价昂贵，并且许多应用可能根本不需要。可以花一些时间去看看新的硬件和新版本

的 MySQL 有哪些变化，或者 MySQL Cluster 有哪些新的进展，甚至去评估一些专门的系统，例如 Clustrix。毕竟数据分片是一个手工搭建的集群系统，如果没有必要，最好不要重复发明轮子。

当存在多个服务器时，可能出现跟一致性或原子性相关的问题。我们看到的最普遍的问题是缺少会话一致性（在网站上发表一篇评论，刷新页面，但找不到刚刚发布的评论），或者无法有效告诉应用哪些服务器是可写的，哪些是可读的。后一种可能更严重，如果
566 将应用的写操作指向多个地方，就会不可避免地遭遇数据问题，需要花费大量时间而且很难解决。负载均衡器可以解决这个问题，但它本身也有一些问题，有时候还会使得原本希望解决的问题恶化。这也是我们在下一章要讲述高可用性的原因。

第 12 章 567

高可用性

本章将讲述我们提到的复制、可扩展性以及高可用性三个主题中的第三个。归根结底，高可用性实际上意味着“更少的宕机时间”。然而糟糕的是，高可用性经常和其他相关的概念混淆，例如冗余、保障数据不丢失，以及负载均衡。我们希望之前的两章已经为清楚地理解高可用性做了足够的铺垫。跟其他两章一样，这一章也不仅仅是关注高可用性的内容，一些相关的话题也会综合阐述。

12.1 什么是高可用性

高可用性实际上有点像神秘的野兽。它通常以百分比表示，这本身也是一种暗示：高可用性不是绝对的，只有相对更高的可用性。100% 的可用性是不可能达到的。可用性的“9”规则是表示可用性目标最普遍的方法。你可能也知道，“5 个 9”表示 99.999% 的正常可用时间。换句话说，每年只允许 5 分钟的宕机时间。对于大多数应用这已经是令人惊叹的数字，尽管还有一些人试图获得更多的“9”。

每个应用对可用性的需求各不相同。在设定一个可用时间的目标之前，先问问自己，是不是确实需要达到这个目标。可用性每提高一点，所花费的成本都会远超之前；可用性的效果和开销的比例并不是线性的。需要保证多少可用时间，取决于能够承担多少成本。高可用性实际上是在宕机造成的损失与降低宕机时间所花费的成本之间取一个平衡。换句话说，如果需要花大量金钱去获得更好的可用时间，但所带来的收益却很低，可能就不值得去做。总的来说，应用在超过一定的点以后追求更高的可用性是非常困难的，成本也会很高，因此我们建议设定一个更现实的目标并且避免过度设计。幸运的是，建立 2 个 9 或 3 个 9 的可用时间的目标可能并不困难，具体情况取决于应用。

568 有时候人们将可用性定义成服务正在运行的时间段。我们认为可用性的定义还应该包括应用是否能以足够好的性能处理请求。有许多方法可以让一个服务器保持运行，但服务并不是真正可用。对一个很大的服务器而言，重启 MySQL 之后，可能需要几个小时才能充分预热以保证查询请求的响应时间是可以接受的，即使服务器只接收了正常流量的一小部分也是如此。

另一个需要考虑的问题是，即使应用并没有停止服务，但是否可能丢失了数据。如果服务器遭遇灾难性故障，可能多少都会丢失一些数据，例如最近已经写入（最新丢失的）二进制日志但尚未传递到备库的中继日志中的事务。你能够容忍吗？大多数应用能够容忍；因为替代方案大多非常昂贵且复杂，或者有一些性能开销。例如，可以使用同步复制，或是将二进制日志放到一个通过 DRBD 进行复制的设备上，这样就算服务器完全失效也不用担心丢失数据。（但是整个数据中心也有可能会掉电。）

一个良好的应用架构通常可以降低可用性方面的需求，至少对部分系统而言是这样的，良好的架构也更容易做到高可用。将应用中重要和不重要的部分进行分离可以节约不少工作量和金钱，因为对于一个更小的系统改进可用性会更容易。可以通过计算“风险敞口（risk exposure)”，将失效概率与失效代价相乘来确认高优先级的风险。画一个简单的风险计算表，以概率、代价和风险敞口作为列，这样很容易找到需要优先处理的项目。

在前一章我们通过讨论如何避免导致糟糕的可扩展性的原因，来推出如何获得更好的可扩展性。这里也会使用相似的方法来讨论可用性，因为我们相信，理解可用性最好的方法就是研究它的反面——宕机时间。接下来的小节我们会讨论为什么会出现宕机。

12.2 导致宕机的原因

我们经常听到导致数据库宕机最主要的原因是编写的 SQL 查询性能很差，真的是这样吗？ 2009 年我们决定分析我们客户的数据库所遇到的问题，以找出那些真正引起宕机的问题，以及如何避免这些问题[注1]。结果证实了一些我们已有的猜想，但也否定了一些(错误的）认识，我们从中学到了很多。

我们首先对宕机事件按表现方式而非导致的原因进行分类。一般来说，“运行环境”是
569 排名第一的宕机类别，大约 35% 的事件属于这一类。运行环境可以看作是支持数据库服务器运行的系统和资源集合，包括操作系统、硬盘以及网络等。性能问题紧随其后，也是约占 35%；然后是复制，占 20%；最后剩下的 10% 包含各种类型的数据丢失或损坏，以及其他问题。

注 1： 我们在一个冗长的白皮书中完整地描述了对客户的宕机事故的分析，并于随后在另一份白皮书中介绍了如何防止宕机，包括可以定期执行的详细检查清单。本书没有这么多篇幅来描述所有的细节，你可以从 Percona 的网站（*http://www.percona.com*）获得这两份白皮书。

我们对事件按类型进行分类后，确定了导致这些事件的原因。以下是一些需要注意的地方：

- 在运行环境的问题中，最普遍的问题是磁盘空间耗尽。
- 在性能问题中，最普遍的宕机原因确实是运行很糟糕的 SQL，但也不一定都是这个原因，比如也有很多问题是由于服务器 Bug 或错误的行为导致的。
- 糟糕的 Schema 和索引设计是第二大影响性能的问题。
- 复制问题通常由于主备数据不一致导致。
- 数据丢失问题通常由于 DROP TABLE 的误操作导致，并总是伴随着缺少可用备份的问题。

复制虽然常被人们用来改善可用时间，但却也可能导致宕机。这主要是由于不正确的使用导致的，即便如此，它也阐明了一个普遍的情况：许多高可用性策略可能会产生反作用，我们会在后面讨论这个话题。

现在我们已经知道了主要宕机类别，以及有什么需要注意，下面我们将专门介绍如何获得高可用性。

12.3 如何实现高可用性

可以通过同时进行以下两步来获得高可用性。首先，可以尝试避免导致宕机的原因来减少宕机时间。许多问题其实很容易避免，例如通过适当的配置、监控，以及规范或安全保障措施来避免人为错误。第二，尽量保证在发生宕机时能够快速恢复。最常见的策略是在系统中制造冗余，并且具备故障转移能力。这两个维度的高可用性可以通过两个相关的度量来确定：平均失效时间（MTBF）和平均恢复时间（MTTR）。一些组织会非常仔细地追踪这些度量值。

第二步——通过冗余快速恢复——很不幸，这里是最应该注意的地方，但预防措施的投资回报率会很高。接下来我们来探讨一些预防措施。

12.3.1 提升平均失效时间（MTBF） 570

其实只要尽职尽责地做好一些应做的事情，就可以避免很多宕机。在分类整理宕机事件并追查导致宕机的根源时，我们还发现，很多宕机本来是有一些方法可以避免的。我们发现大部分宕机事件都可以通过全面的常识性系统管理办法来避免。以下是从我们的白皮书中摘录的指导性建议，在白皮书中有我们详细的分析结果。

- 测试恢复工具和流程，包括从备份中恢复数据。
- 遵从最小权限原则。
- 保持系统干净、整洁。
- 使用好的命名和组织约定来避免产生混乱，例如服务器是用于开发还是用于生产环境。
- 谨慎安排升级数据库服务器。
- 在升级前，使用诸如 Percona Toolkit 中的 *pt-upgrade* 之类的工具仔细检查系统。
- 使用 InnoDB 并进行适当的配置，确保 InnoDB 是默认存储引擎。如果存储引擎被禁止，服务器就无法启动。
- 确认基本的服务器配置是正确的。
- 通过 `skip_name_resolve` 禁止 DNS。
- 除非能证明有效，否则禁用查询缓存。
- 避免使用复杂的特性，例如复制过滤和触发器，除非确实需要。
- 监控重要的组件和功能，特别是像磁盘空间和 RAID 卷状态这样的关键项目，但也要避免误报，只有当确实发生问题时才发送告警。
- 尽量记录服务器的状态和性能指数，如果可能就尽量久地保存。
- 定期检查复制完整性。
- 将备库设置为只读，不要让复制自动启动。
- 定期进行查询语句审查。
- 归档并清理不需要的数据。
- 为文件系统保留一些空间。在 GNU/Linux 中，可以使用 *-m* 选项来为文件系统本身保留空间。还可以在 LVM 卷组中留下一些空闲空间。或者，更简单的方法，仅仅创建一个巨大的空文件，在文件系统快满时，直接将其删除。[注2]
- 571 养成习惯，评估和管理系统的改变、状态以及性能信息。

我们发现对系统变更管理的缺失是所有导致宕机的事件中最普遍的原因。典型的错误包括粗心的升级导致升级失败并遭遇一些 Bug，或是尚未测试就将 Schema 或查询语句的更改直接运行到线上，或者没有为一些失败的情况制定计划，例如达到了磁盘容量限制。另外一个导致问题的主要原因是缺少严格的评估，例如因为疏忽没有确认备份是否是可以恢复的。最后，可能没有正确地监控 MySQL 的相关信息。例如缓存命中率报警并不能说明出现问题，并且可能产生大量的误报，这会使监控系统被认为不太有用，于是一些人就会忽略报警。有时候监控系统失效了，甚至没人会注意到，直至你的老板质问你，“为什么 Nagios 没有告诉我们磁盘已经满了”。

注 2： 这是 100% 跨平台兼容的。

12.3.2 降低平均恢复时间（MTTR）

之前提到，可以通过减少恢复时间来获得高可用性。事实上，一些人走得更远，只专注于减少恢复时间的某个方面：通过在系统中建立冗余来避免系统完全失效，并避免单点失效问题。

在降低恢复时间上进行投资非常重要，一个能够提供冗余和故障转移能力的系统架构，则是降低恢复时间的关键环节。但实现高可用性不单单是一个技术问题，还有许多个人和组织的因素。组织和个人在避免宕机和从宕机事件中恢复的成熟度和能力层次各不相同。

团队成员是最重要的高可用性资产，所以为恢复制定一个好的流程非常重要。拥有熟练技能、应变能力、训练有素的雇员，以及处理紧急事件的详细文档和经过仔细测试的流程，对从宕机中恢复有巨大的作用。但也不能完全依赖工具和系统，因为它们并不能理解实际情况的细微差别，有时候它们的行为在一般情况下是正确的，但在某些场景下却会是个灾难！

对宕机事件进行评估有助于提升组织学习能力，可以帮助避免未来发生相似的错误，但是不要对“事后反思”或“事后的调查分析”期待太高。后见之明被严重曲解，并且一味想找到导致问题的唯一根源，这可能会影响你的判断力[注3]。许多流行的方法，例如“五个为什么”，可能会被过度使用，导致一些人将他们的精力集中在找到唯一的替罪羊。很难去回顾我们解决的问题当时所处的状况，也很难理解真正的原因，因为原因通常是多方面的。因此，尽管事后反思可能是有用的，但也应该对结论有所保留。即使是我们给出的建议，也是基于长期研究导致宕机事件的原因以及如何预防它们所得，并且只是我们的观点而已。 572

这里我们要反复提醒：所有的宕机事件都是由多方面的失效联合在一起导致的。因此，可以通过利用合适的方法确保单点的安全来避免。整个链条必须要打断，而不仅仅是单个环节。例如，那些向我们求助恢复数据的人不仅遭受数据丢失（存储失效，DBA 误操作等），同时还缺少一个可用的备份。

这样说来，当开始调查并尝试阻止失效或加速恢复时，大多数人和组织不应太过于内疚，而是要专注于技术上的一些措施——特别是那些很酷的方法，例如集群系统和冗余架构。这些是有用的，但要记住这些系统依然会失效。事实上，在本书第二版中提到的 MMM 复制管理，我们已经失去了兴趣，因为它被证明可能导致更多的宕机时间。你应该不会

注 3：这里推荐两篇反驳常识的文章：Richard Cook 的论文“How Complex Systems Fail”（*http://www.ctlab.org/documents/How%20Complex%20Systems%20Fail.pdf*）和 Malcolm Gladwell 在他的 *What the Dog Saw*（Little, Brown）一书中关于挑战者号航天飞机灾难事件的文章。

奇怪一组 Perl 脚本会陷于混乱，但即使是特别昂贵并精密设计的系统也会出现灾难性的失效——是的，即使是花费了大量金钱的 SAN 也是如此。我们已经见过太多的 SAN 失效。

12.4 避免单点失效

找到并消除系统中的可能失效的单点，并结合切换到备用组件的机制，这是一种通过减少恢复时间（MTTR）来改善可用性的方法。如果你够聪明，有时候甚至能将实际的恢复时间降低至 0，但总的来说这很困难。（即使一些非常引人注目的技术，例如昂贵的负载均衡器，在发现问题并进行反馈时也会导致一定的延迟。）

思考并梳理整个应用，尝试去定位任何可能失效的单点。是一个硬盘驱动器，一台服务器，一台交换或路由器，还是某个机架的电源？所有数据都在一个数据中心，或者冗余数据中心是由同一个公司提供的吗？系统中任何不冗余的部分都是一个可能失效的单点。其他比较普遍的单点失效依赖于一些服务，例如 DNS、单一网络提供商[注4]、单个云“可用区域”，以及单个电力输送网，具体有哪些取决于你的关注点。

单点失效并不总是能够消除。增加冗余或许也无法做到，因为有些限制无法避开，例如地理位置，预算，或者时间限制等。试着去理解每一个影响可用性的部分，采取一种平衡的观点来看待风险，并首先解决其中影响最大的那个。一些人试图编写一个软件来处
573 理所有的硬件失效，但软件本身导致的宕机时间可能比它节约的还要多。也有人想建立一种“永不沉没”的系统，包括各种冗余，但他们忘记了数据中心可能掉电或失去连接。或许他们彻底忘记了恶意攻击者和程序错误的可能性，这些情况可能会删除或损坏数据——一个不小心执行的 DROP TABLE 也会产生宕机时间。

可以采用两种方法来为系统增加冗余：增加空余容量和重复组件。增加容量余量通常很简单——可以使用本章或前一章讨论的任何技术。一个提升可用性的方法是创建一个集群或服务器池，并使用负载均衡解决方案。如果一台服务器失效，其他服务器可以接管它的负载。有些人有意识地不使用组件的全部能力，这样可以保留一些“动态余量”来处理因为负载增加或组件失效导致的性能问题。

出于很多方面的考虑会需要冗余组件，并在主要组件失效时能有一个备件来随时替换。冗余组件可以是空闲的网卡、路由器或者硬盘驱动器——任何能想到的可能失效的东西。完全冗余 MySQL 服务器可能有点困难，因为一个服务器在没有数据时毫无用处。这意味着你必须确保备用服务器能够获得主服务器上的数据。共享或复制存储是一个比较流行的办法，但这真的是一个高可用性架构吗？让我们深入其中看看。

注 4： 感觉太偏执了？检查你的冗余网络连接是不是真的连接到不同的互联网主干，确保它们的物理位置不在同一条街道或者同一个电线杆上，这样它们才不会被同一个挖土机或者汽车破坏掉。

12.4.1 共享存储或磁盘复制

共享存储能够为数据库服务器和存储解耦合，通常使用的是 SAN。使用共享存储时，服务器能够正常挂载文件系统并进行操作。如果服务器挂了，备用服务器可以挂载相同的文件系统，执行需要的恢复操作，并在失效服务器的数据上启动 MySQL。这个过程在逻辑上跟修复那台故障的服务器没什么两样，不过更快速，因为备用服务器已经启动，随时可以运行。当开始故障转移时，检查文件系统、恢复 InnoDB 以及预热[注5]是最有可能遇到延迟的地方，但检测失效本身在许多设置中也会花费很长时间。

共享存储有两个优点：可以避免除存储外的其他任何组件失效所引起的数据丢失，并为非存储组件建立冗余提供可能。因此它有助于减少系统一些部分的可用性需求，这样就可以集中精力关注一小部分组件来获得高可用性。不过，共享存储本身仍是可能失效的单点。如果共享存储失效了，那整个系统也失效了，尽管 SAN 通常设计良好，但也可能失效，有时候需要特别关注。就算 SAN 本身拥有冗余也会失效。 574

> **主动—主动访问模式的共享存储怎么样？**
>
> 在一个 SAN、NAS 或者集群文件系统上以主动—主动模式运行多个实例怎么样？MySQL 不能这么做。因为 MySQL 并没有被设计成和其他 MySQL 实例同步对数据的访问，所以无法在同一份数据上开启多个 MySQL 实例。（如果在一份只读的静态数据上使用 MyISAM，技术上是可行的，但我们还没有见过任何实际的应用。）[注6]
>
> MySQL 的一个名为 ScaleDB 的存储引擎在底层提供了操作共享存储的 API，但我们还没有评估过，也没有见过任何生产环境使用。在写作本书时它还是 beta 版。

共享存储本身也有风险，如果 MySQL 崩溃等故障导致数据文件损坏，可能会导致备用服务器无法恢复。我们强烈建议在使用共享存储策略时选择 InnoDB 存储引擎或其他稳定的 ACID 存储引擎。一次崩溃几乎肯定会损坏 MyISAM 表，需要花费很长时间来修复，并且会丢失数据。我们也强烈建议使用日志型文件系统。我们见过比较严重的情况是，使用非日志型文件系统和 SAN（这是文件系统的问题，跟 SAN 无关）导致数据损坏无法恢复。

注 5： Percona Server 提供了一个新特性，能够把 buffer pool 保存下来并在重启后还原，在使用共享存储时能够很好地工作。这可以减少几个小时甚至好几天的预热时间。MySQL 5.6 也有相似的特性。

注 6： MySQL 5.6.8 之后 InnoDB 也增加了一个只读模式，可以只读的方式用多个实例访问一份只读数据文件。——译者注

磁盘复制技术是另外一个获得跟 SAN 类似效果的方法。MySQL 中最普遍使用的磁盘复制技术是 DRBD（*http://www.drbd.org*），并结合 Linux-HA 项目中的工具使用（后面会介绍到）。

DRBD 是一个以 Linux 内核模块方式实现的块级别同步复制技术。它通过网卡将主服务器的每个块复制到另外一个服务器的块设备上（备用设备），并在主设备提交块之前记录下来[注7]。由于在备用 DRBD 设备上的写入必须要在主设备上的写入完成之前，因此备用设备的性能至少要和主设备一样，否则就会限制主设备的写入性能。同样，如果正在使用 DRBD 磁盘复制技术以保证在主设备失效时有一个可随时替换的备用设备，备用服务器的硬件应该跟主服务器的相匹配。带电池写缓存的 RAID 控制器对 DRBD 而言几乎是必需的，因为在没有这样的控制器时性能可能会很差。

如果主服务器失效，可以把备用设备提升为主设备。因为 DRBD 是在磁盘块层进行复制，而文件系统也可能会不一致。这意味着最好是使用日志型文件系统来做快速恢复。一旦
575 设备恢复完成，MySQL 还需要运行自身的恢复。原故障服务器恢复后，会与新的主设备进行同步，并假定自身角色为备用设备。

从如何实际地实现故障转移的角度来看，DRBD 和 SAN 很相似：有一个热备机器，开始提供服务时会使用和故障机器相同的数据。最大的不同是，DRBD 是复制存储——不是共享存储——所以当使用 DRBD 时，获得的是一份复制的数据，而 SAN 则是使用与故障机器同一物理设备上的相同数据副本。换句话说，磁盘复制技术的数据是冗余的，所以存储和数据本身都不会存在单点失效问题。这两种情况下，当启动备用机器时，MySQL 服务器的缓存都是空的。相比之下，备库的缓存至少是部分预热的。

DRBD 有一些很好的特性和功能，可以防止集群软件普遍会遇到的一些问题。一个典型的例子是“脑裂综合征”，在两个节点同时提升自己为主服务器时会发生这种问题。可以通过配置 DRBD 来防止这种事件发生。但是 DRBD 也不是一个能满足所有需求的完美解决方案。我们来看看它有哪些缺点：

- DRBD 的故障转移无法做到秒级以内。它通常至少需要几秒钟时间来将备用设备提升成主设备，这还不包括任何必要的文件系统恢复和 MySQL 恢复。
- 它很昂贵，因为必须在主动—被动模式下运行。热备服务器的复制设备因为处于被动模式，无法用于其他任务。当然这是不是缺点取决于看问题的角度。如果你希望获得真正的高可用性并且在发生故障时不能容忍服务降级，就不应该在一台机器上运行两台服务器的负载量，因为如果这么做了，当其中一台发生故障时，就无法处

注 7：　事实上可以调整 DRDB 的同步级别，将其设置成异步等待远程设备接收数据，或者在远程设备将数据写入磁盘前一直阻塞住。同样，强烈建议为 DRBD 专门使用一块网卡。

理这些负载了。可以用这些备用服务器做一些其他用途，例如用作备库，但还是会有一些资源浪费。

- 对于 MyISAM 表实际上用处不大，因为 MyISAM 表崩溃后需要花费很长时间来检查和修复。对任何期望获得高可用性的系统而言，MyISAM 都不是一个好选择；请使用 InnoDB 或其他支持快速、安全恢复的存储引擎来代替 MyISAM。
- DRBD 无法代替备份。如果磁盘由于蓄意的破坏、误操作、Bug 或者其他硬件故障导致数据损坏，DRBD 将无济于事。此时复制的数据只是被损坏数据的完美副本。你需要使用备份（或 MySQL 延时复制）来避免这些问题。
- 对写操作而言增加了负担。具体会增加多少负担呢？通常可以使用百分比来表示，但这并不是一个好的度量方法。你需要理解写入时增加的延迟主要由网络往返开销和远程服务器存储导致，特别是对于小的写入而言延迟会更大。尽管增加的延迟可能也就 0.3ms，这看起来比在本地磁盘上 I/O 的 4 ~ 10ms 的延迟要小很多，但却是正常的带有写缓存的 RAID 控制器的延迟的 3 ~ 4 倍。使用 DRBD 导致服务器变慢 576
最常见的原因是 MySQL 使用 InnoDB 并采取了完全持久化模式[注8]，这会导致许多小的写入和 fsync() 调用，通过 DRBD 同步时会非常慢。[注9]

我们倾向于只使用 DRBD 复制存放二进制日志的设备。如果主动节点失效，可以在被动节点上开启一个日志服务器，然后对失效主库的所有备库应用这些二进制日志。接下来可以选择其中一个备库提升为主库，以代替失效的系统。

说到底，共享存储和磁盘复制与其说是高可用性（低宕机时间）解决方案，不如说是一种保证数据安全的方法。只要拥有数据，就可以从故障中恢复，并且比无法恢复的情况的 MTTR 更低。（即使是很长的恢复时间也比不能恢复要快。）但是相比于备用服务器启动并一直运行的架构，大多数共享存储或磁盘复制架构会增加 MTTR。有两种启用备用设备并运行的方法：我们在第 10 章讨论的标准的 MySQL 复制，以及接下来会讨论的同步复制。

12.4.2 MySQL 同步复制

当使用同步复制时，主库上的事务只有在至少一个备库上提交后才能认为其执行完成。这实现了两个目标：当服务器崩溃时没有提交的事务会丢失，并且至少有一个备库拥有实时的数据副本。大多数同步复制架构运行在主动 - 主动模式。这意味着每个服务器在任何时候都是故障转移的候选者，这使得通过冗余获得高可用性更加容易。

注 8： 这里的意思应该是 innodb_flush_log_at_trx_commit=1 的情况。——译者注

注 9： 另一方面，大的序列写入又是另外一种情况，由 DRBD 导致的增加的延迟实际上消失了，但吞吐量的限制依然存在。一个合适的 RAID 阵列能够提供 200 ~ 500MB/s 的序列写入吞吐，大大超过千兆网络所能获得的吞吐量。

在写作本书时，MySQL 本身并不支持同步复制注 10，但有两个基于 MySQL 的集群解决方案支持同步复制。你还可以阅读第 10 章、第 11 章和第 13 章讨论的其他产品，例如 Continuent Tungsten 以及 Clustrix，这些都相当有意思。

1. MySQL Cluster

577 MySQL 中的同步复制首先出现在 MySQL Cluster（NDB Cluster）。它在所有节点上进行同步的主 - 主复制。这意味着可以在任何节点上写入；这些节点拥有等同的读写能力。每一行都是冗余存储的，这样即使丢失了一个节点，也不会丢失数据，并且集群仍然能提供服务。尽管 MySQL Cluster 还不是适用于所有应用的完美解决方案，但正如我们在前一章提到的，在最近的版本中它做了非常快速的改进，现在已经拥有大量的新特性和功能：非索引数据的磁盘存储、增加数据节点能够在线扩展、使用 ndbinfo 表来管理集群、配置和管理集群的脚本、多线程操作、下推（push-down）的关联操作（现在称为自适应查询本地化）、能够处理 BLOB 列和很多列的表、集中式的用户管理，以及通过像 memcached 协议一样的 NDB API 来实现 NoSQL 访问。在下一个版本中将包含最终一致运行模式，包括为跨数据中心的主动 - 主动复制提供事务冲突检测和跨 WAN 解决方案。简而言之，MySQL Cluster 是一项引人注目的技术。

现在至少有两个为简化集群部署和管理提供附加产品的供应商：Oracle 针对 MySQL Cluster 的服务支持包含了 MySQL Cluster Manager 工具；Severalnines 提供了 Cluster Control 工具（*http://www.severalnines.com*），该工具还能够帮助部署和管理复制集群。

2. Percona XtraDB Cluster

Percona XtraDB Cluster 是一个相对比较新的技术，基于已有的 XtraDB（InnoDB）存储引擎增加了同步复制和集群特性，而不是通过一个新的存储引擎或外部服务器来实现。它是基于 Galera（支持在集群中跨节点复制写操作）实现的注 11，这是一个在集群中不同节点复制写操作的库。跟 MySQL Cluster 类似，Percona XtraDB Cluster 提供同步多主库复制注 12，支持真正的任意节点写入能力，能够在节点失效时保证数据零丢失（持久性，ACID 中的 D），另外还提供高可用性，在整个集群没有失效的情况下，就算单个节点失效也没有关系。

Galera 作为底层技术，使用一种被称为写入集合（write-set）复制的技术。写入集合实际上被作为基于行的二进制日志事件进行编码，目的是在集群中的节点间传输并进行更

注 10： MySQL 5.5 支持半同步复制，参见第 10 章。

注 11： Galera 技术由 Codership Oy（*http://www.codership.com*）开发，可以作为一个补丁在标准的 MySQL 和 InnoDB 中使用。Percona XtraDB Cluster 除了其他特性和功能外，还包含这组补丁的修改版本。Percona XtraDB Cluster 是一个可以直接使用的基于 Galera 的解决方案。

注 12： 你可以通过配置主备只写入其中一个节点来实现，但在集群配置中，对于这种模式的操作没有什么不同。

新，但是这不要求二进制日志是打开的。

Percona XtraDB Cluster 的速度很快。跨节点复制实际上比没有集群还要快，因为在完全持久性模式下，写入远程 RAM 比写入本地磁盘要快。如果你愿意，可以选择通过降低每个节点的持久性来获得更好的性能，并且可以依赖于多个节点上的数据副本来获得持久性。NDB 也是基于同样的原理实现的。集群在整体上的持久性并没有降低；仅仅是降低了本地节点的持久性。除此之外，还支持行级别的并发（多线程）复制，这样就可以利用多个 CPU 核心来执行写入集合。这些特性结合起来使得 Percona XtraDB Cluster 非常适合云计算环境，因为云计算环境中的 CPU 和磁盘通常比较慢。 578

在集群中通过设置 auto_increment_offset 和 auto_increment_increment 来实现自增键，以使节点间不会生成冲突的主键值。锁机制和标准 InnoDB 完全相同，使用的是乐观并发控制。当事务提交时，所有的更新是序列化的，并在节点间传输，同时还有一个检测过程，以保证一旦发生更新冲突，其中一些更新操作需要丢弃。这样如果许多节点同时修改同样的数据，可能产生大量的死锁和回滚。

Percona XtraDB Cluster 只要集群内在线的节点数不少于"法定人数（quorum）"就能保证服务的高可用性。如果发现某个节点不属于"法定人数"中的一员，就会从集群中将其踢出。被踢出的节点在再次加入集群前必须重新同步。因此集群也无法处理"脑裂综合征"；如果出现脑裂则集群会停止服务。在一个只有两个节点的集群中，如果其中一个节点失效，剩下的一个节点达不到"法定人数"，集群将停止服务，所以实际上最少需要三个节点才能实现高可用的集群。

Percona XtraDB Cluster 有许多优点：

- 提供了基于 InnoDB 的透明集群，所以无须转换到另外的技术，例如 NDB 这样完全不同的技术需要很多学习成本和管理。
- 提供了真正的高可用性，所有节点等效，并在任何时候提供读写服务。相比较而言，MySQL 内建的异步复制和半同步复制必须要有一个主库，并且不能保证数据被复制到备库，也无法保证备库数据是最新的并能够随时提升为主库。
- 节点失效时保证数据不丢失。实际上，由于所有的节点都拥有全部数据，因此可以丢失任意一个节点而不会丢失数据(即使集群出现脑裂并停止工作)。这和NDB不同，NDB 通过节点组进行分区，当在一个节点组中的所有服务器失效时就可能丢失数据。
- 备库不会延迟，因为在事务提交前，写入集合已经在集群的所有节点上传播并被确认了。
- 因为是使用基于行的日志事件在备库上进行更新，所以执行写入集合比直接执行更新的开销要小很多，就和使用基于行的复制差不多。当结合多线程应用的写入集合时，

可以使其比 MySQL 本身的复制更具备可扩展性。

579 当然我们也需要提及 Percona XtraDB Cluster 的一些缺点：

- 它很新，因此还没有足够的经验来证明其优点和缺点，也缺乏合适的使用案例。
- 整个集群的写入速度由最差的节点决定。因此所有的节点最好拥有相同的硬件配置，如果一个节点慢下来（例如，RAID 卡做了一次 battery-learn 循环），所有的节点都会慢下来。如果一个节点接收写入操作变慢的可能性为 P，那么有 3 个节点的集群变慢的可能性为 $3P$。
- 没有 NDB 那样节省空间，因为每个节点都需要保存全部数据，而不是仅仅一部分。但另一方面，它基于 Percona XtraDB（InnoDB 的增强版本），也就没有 NDB 关于磁盘数据限制的担忧。
- 当前不支持一些在异步复制中可以做的操作，例如在备库上离线修改 schema，然后将其提升为主库，然后在其他节点上重复离线修改操作。当前可替代的选择是使用诸如 Percona Toolkit 中的在线 schema 修改工具。不过滚动式 schema 升级（rolling schema upgrade）在写作本书时也即将发布。
- 当向集群中增加一个新节点时，需要复制所有的数据，还需要跟上不断进行的写入操作，所以一个拥有大量写入的大型集群很难进行扩容。这实际上限制了集群的数据大小。我们无法确定具体的数据。但悲观地估计可能低至 100GB 或更小，也可能会大得多。这一点需要时间和经验来证明。
- 复制协议在写入时对网络波动比较敏感，这可能导致节点停止并从集群中踢出。所以我们推荐使用高性能网络，另外还需要很好的冗余。如果没有可靠的网络，可能会导致需要频繁地将节点加入到集群中。这需要重新同步数据。在写本书时，有一个几乎接近可用的特性，即通过增量状态传输来避免完全复制数据集，因此未来这并不是一个问题。还可以配置 Galera 以容忍更大的网络延迟（以延迟故障检测为代价），另外更加可靠的算法也计划在未来的版本中实现。
- 如果没有仔细关注，集群可能会增长得太大，以至于无法重启失效节点，就像在一个合理的时间范围内，如果在日常工作中没有定期做恢复演练，备份也会变得太过庞大而无法用于恢复。我们需要更多的实践经验来了解它事实上是如何工作的。
- 由于在事务提交时需要进行跨节点通信，写入会更慢，随着集群中增加的节点越来越多，死锁和回滚也会更加频繁。（参阅前一章了解为什么会发生这种情况。）

Percona XtraDB Cluster 和 Galera 都处于其生命周期的早期，正在被快速地修改和改进。
580 在写作本书时，正在进行或即将进行的改进包括群体行为、安全性、同步性、内存管理、状态转移等。未来还可以为离线节点执行诸如滚动式 schema 变更的操作。

12.4.3 基于复制的冗余

复制管理器是使用标准 MySQL 复制来创建冗余的工具[注13]。尽管可以通过复制来改善可用性，但也有一些“玻璃天花板”会阻止 MySQL 当前版本的异步复制和半同步复制获得和真正的同步复制相同的结果。复制无法保证实时的故障转移和数据零丢失，也无法将所有节点等同对待。

复制管理器通常监控和管理三件事：应用和 MySQL 间的通信、MySQL 服务器的健康度，以及 MySQL 服务器间的复制关系。它们既可以修改负载均衡的配置，也可以在必要的时候转移虚拟 IP 地址以使应用连接到合适的服务器上，还能够在一个伪集群中操纵复制以选择一个服务器作为写入节点。大体上操作并不复杂：只需要确定写入不会发送到一个还没有准备好提供写服务的服务器上，并保证当需要提升一台备库为主库时记录下正确的复制坐标。

这听起来在理论上是可行的，但我们的经验表明实际上并不总是能有效工作。事实上这非常糟糕，有些时候最好有一些轻量级的工具集来帮助从常见的故障中恢复并以很少的开销获得较高的可用性。不幸的是，在写作本书时我们还没有听说任何一个好的工具集可以可靠地完成这一点。稍后我们会介绍两个复制管理器[注14]，其中一个很新，而另外一个则有很多问题。

我们发现很多人试图去写自己的复制管理器。他们常常会陷入很多人已经遭遇过的陷阱。自己去写一个复制管理器并不是好主意。异步组件有大量的故障形式，很多你从未亲身经历过，其中一些甚至无法理解，并且程序也无法适当处理，因此从这些异步组件中得到正确的行为相当困难，并且可能遭遇数据丢失的危险。事实上，机器刚开始出现问题时，由一个经验丰富的人来解决是很快的，但如果其他人做了一些错误的修复操作则可能导致问题更严重。

我们要提到的第一个复制管理器是 MMM（*http://mysql-mmm.org*），本书的作者对于该工具集是否适用于生产环境部署的意见并不一致（尽管该工具的原作者也承认它并不可靠）。我们中有些人认为它在一些人工—故障转移模式下的场景中比较有用，而有些人甚至从不使用这个工具。我们的许多客户在自动—故障转移模式下使用该工具时确实遇到了许多严重的问题。它会导致健康的服务器离线，也可能将写入发送到错误的地点，并将备库移动到错误的坐标。有时混乱就接踵而至。 581

另外一个比较新一点的工具是 Yoshinori Matsunobu 的 MHA 工具集（*http://code.google.com/p/mysql-master-ha/*）。它和 MMM 一样是一组脚本，使用相同的通用技术来建

注 13： 在本小节我们会很小心，以避免产生混淆。冗余并不等同于高可用性。

注 14： 我们同样在开发基于 Pacemaker 和 Linux-HA 栈的解决方案，但并不准备在本书中提及。这个脚注稍后会自毁，10……9……8……

立一个伪集群，但它不是一个完全的替换者；它不会去做太多的事情，并且依赖于 Pacemaker 来转移虚拟 IP 地址。一个主要的不同是，MHA 有一个很好的测试集，可以防止一些 MMM 遇到过的问题。除此之外，我们对该工具集还没有更多的认识，我们只和 Yoshinori 讨论过，但还没有真正使用过。

基于复制的冗余最终来说好坏参半。只有在可用性的重要性远比一致性或数据零丢失保证更重要时才推荐使用。例如，一些人并不会真的从他们的网站功能中获利，而是从它的可用性中赚钱。谁会在乎是否出现了故障导致一张照片丢失了几条评论或其他什么东西呢？只要广告收益继续滚滚而来，可能并不值得花更多成本去实现真正的高可用性。但还是可以通过复制来建立“尽可能的”高可用性，当遇到一些很难处理的严重宕机时可能会有所帮助。这是一个大赌注，并且可能对大多数人而言太过于冒险，除非是那些老成（或者专业）的用户。

问题是许多用户不知道如何去证明自己有资格并评估复制“轮盘赌”是否适合他们。这有两个方面的原因。第一，他们并没有看到“玻璃天花板”，错误地认为一组虚拟 IP 地址、复制以及管理脚本能够实现真正的高可用性。第二，他们低估了技术的复杂度，因此也低估了严重故障发生后从中恢复的难度。一些人认为他们能够使用基于复制的冗余技术，但随后他们可能会更希望选择一个有更强保障的简单系统。

其他一些类型的复制，例如 DRBD 或者 SAN，也有它们的缺点——请不要认为我们将这些技术说得无所不能而把 MySQL 自身的复制贬得一团糟，那不是我们的本意。你可以为 DRBD 写出低质量的故障转移脚本，这很简单，就像为 MySQL 复制编写脚本一样。主要的区别是 MySQL 复制非常复杂，有很多非常细小的差别，并且不会阻止你干坏事。

12.5 故障转移和故障恢复

冗余是很好的技术，但实际上只有在遇到故障需要恢复时才会用到。（见鬼，这可以用备份来实现）。冗余一点儿也不会增加可用性或减少宕机。在故障转移的过程中，高可用性
582 是建立在冗余的基础上。当有一个组件失效，但存在冗余时，可以停止使用发生故障的组件，而使用冗余备件。冗余和故障转移结合可以帮助更快地恢复，如你所知，MTTR 的减少将降低宕机时间并改善可用性。

在继续这个话题之前，我们先来定义一些术语。我们统一使用“故障转移（failover）”，有些人使用“回退”（fallback）表达同一意思。有时候也有人说“切换（switchover）”，以表明一次计划中的切换而不是故障后的应对措施。我们也会使用“故障恢复”来表示故障转移的反面。如果系统拥有故障恢复能力，故障转移就是一个双向过程：当服务器 A 失效，服务器 B 代替它，在修复服务器 A 后可以再替换回来。

故障转移比仅仅从故障中恢复更好。也可以针对一些情况制订故障转移计划，例如升级、schema 变更、应用修改，或者定期维护，当发生故障时可以根据计划进行故障转移来减少宕机时间（改善可用性）。

你需要确定故障转移到底需要多快，也要知道在一次故障转移后替换一个失效组件应该多快。在你恢复系统耗尽的备件容量之前，会出现冗余不足，并面临额外风险。因此，拥有一个备件并不能消除即时替换失效组件的需求。构建一个新的备用服务器，安装操作系统，并复制数据的最新副本，可以多快呢？有足够的备用机器吗？你可能需要不止一台以上。

故障转移的缘由各不相同。我们已经讨论了其中的一些，因为负载均衡和故障转移在很多方面很相似，它们之间的分界线比较模糊。总的来说，我们认为一个完全的故障转移解决方案至少能够监控并自动替换组件。它对应用应该是透明的。负载均衡不需要提供这些功能。

在 UNIX 领域，故障转移常常使用 High Availability Linux 项目（*http://linux-ha.org*）提供的工具来完成，该项目可在许多类 UNIX 系统上运行，而不仅仅是 Linux。Linux-HA 栈在最近几年明显多了很多新特性。现在大多数人认为 Pacemaker 是栈中的一个主要组件。Pacemaker 替代了老的心跳工具。还有其他一些工具实现了 IP 托管和负载均衡功能。可以将它们跟 DRBD 和 / 或者 LVS 结合起来使用。

故障转移最重要的部分就是故障恢复。如果服务器间不能自如切换，故障转移就是一个死胡同，只能是延缓宕机时间而已。这也是我们倾向于对称复制布局，例如双主配置，而不会选择使用三台或更多的联合主库（co-master）来进行环形复制的原因。如果配置是对等的，故障转移和故障恢复就是在相反方向上的相同操作。（值得一提的是 DRBD 具有内建的故障恢复功能。）

在一些应用中，故障转移和故障恢复需要尽量快速并具备原子性。即便这不是决定性的，不依靠那些不受你控制的东西也依然是个好主意，例如 DNS 变更或者应用程序配置文件。 583
一些问题直到系统变得更加庞大时才会显现出来，例如当应用程序强制重启以及原子性需求出现时。

由于负载均衡和故障转移两者联系较紧密，有些硬件和软件是同时为这两个目的设计的，因此我们建议所选择的任何负载均衡技术应该都提供故障转移功能。这也是我们建议避免使用 DNS 和修改代码来做负载均衡的真实原因。如果为负载均衡采用了这些策略，就需要做一些额外的工作：当需要高可用性时，不得不重写受影响的代码。

以下小节讨论了一些比较普遍的故障转移技术。可以手动执行或使用工具来实现。

12.5.1 提升备库或切换角色

提升一台备库为主库，或者在一个主—主复制结构中调换主动和被动角色，这些都是许多 MySQL 故障转移策略很重要的一部分。具体细节参见第 10 章。正如本章之前提到的，我们不能认定自动化工具总能在所有的情况下做正确的事情——或者至少以我们的名誉担保没有这样的工具。

你不应该假定在发生故障时能够立刻切换到被动备库，这要看具体的工作负载。备库会重放主库的写入，但如果不用来提供读操作，就无法进行预热来为生产环境负载提供服务。如果希望有一个随时能承担读负载的备库，就要不断地"训练"它，既可以将其用于分担工作负载，也可以将生产环境的读查询镜像到备库上。我们有时候通过监听 TCP 流量，截取出其中的 SELECT 查询，然后在备库上重放来实现这个目的。Percona Toolkit 中有一些工具可以做到这一点。

12.5.2 虚拟 IP 地址或 IP 接管

可以为需要提供特定服务的 MySQL 实例指定一个逻辑 IP 地址。当 MySQL 实例失效时，可以将 IP 地址转移到另一台 MySQL 服务器上。这和我们在前一章提到的思想本质上是相同的，唯一的不同是现在是用于故障转移，而不是负载均衡。

这种方法的好处是对应用透明。它会中断已有的连接，但不要求修改配置。有时候还可以原子地转移 IP 地址，保证所有的应用在同一时间看到这一变更。当服务器在可用和不可用状态间"摇摆"时，这一点尤其重要。

584 以下是它的一些不足之处：

- 需要把所有的 IP 地址定义在同一网段，或者使用网络桥接。
- 改变 IP 地址需要系统 root 权限。
- 有时候还需要更新 ARP 缓存。有些网络设备可能会把 ARP 信息保存太久，以致无法即时将一个 IP 地址切换到另一个 MAC 地址上。我们看到过很多网络设备或其他组件不配合切换的例子，结果系统的许多部分可能无法确定 IP 地址到底在哪里。
- 需要确定网络硬件支持快速 IP 接管。有些硬件需要克隆 MAC 地址后才能工作。
- 有些服务器即使完全丧失功能也会保持持有 IP 地址，所以可能需要从物理上关闭或断开网络连接。这就是为人所熟知的"击中其他节点的头部"(shoot the other node in the head，简称 STONITH)。它还有一个更加微妙并且比较官方的名字：击剑(fencing)。

浮动 IP 地址和 IP 接管能够很好地应付彼此临近（也就是在同一子网内）的机器之间的

故障转移。但是最后需要提醒的是，这种策略并不总是万无一失，还取决于网络硬件等因素。

> **等待更新扩散**
>
> 经常有这种情况，在某一层定义了一个冗余后，需要等待低层执行一些改变。在本章前面的篇幅里，我们指出通过 DNS 修改服务器是一个很脆弱的解决方案，因为 DNS 的更新扩散速度很慢，改变 IP 地址可给予你更多的控制，但在一个 LAN 中的 IP 地址同样依赖于更低层——ARP——来扩散更新。

12.5.3 中间件解决方案

可以使用代理、端口转发、网络地址转换（NAT）或者硬件负载均衡来实现故障转移和故障恢复。这些都是很好的解决方案，不像其他方法可能会引入一些不确定性（所有系统组件认同哪一个是主库吗？它能够及时并原子地更改吗？），它们是控制应用和服务器间连接的中枢。但是，它们自身也引入了单点失效，需要准备冗余来避免这个问题。

使用这样的解决方案，你可以将一个远程数据中心设置成看起来好像和应用在同一个网络里。这样就可以使用诸如浮动 IP 地址这样的技术让应用和一个完全不同的数据中心开始通信。你可以配置每个数据中心的每台应用服务器，通过它自己的中间件连接，将流量路由到活跃数据中心的机器上。图 12-1 描述了这种配置。 585

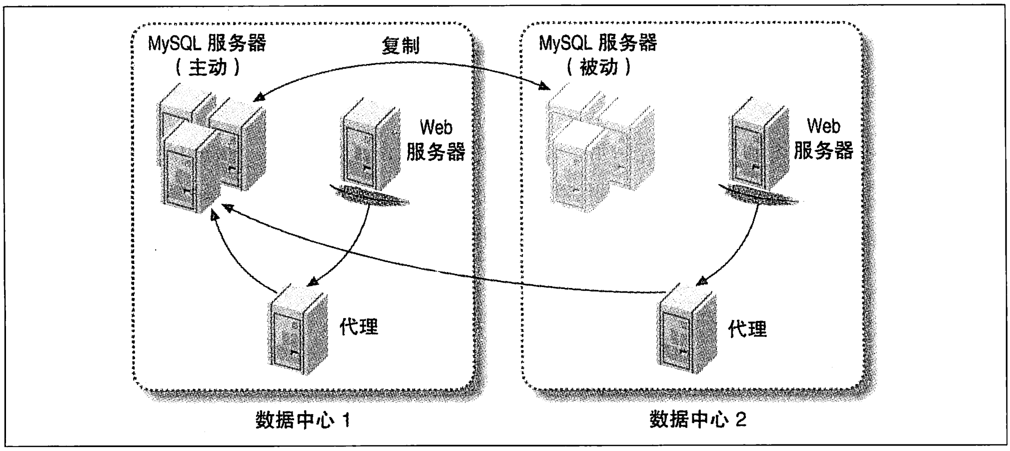


图12-1：使用中间件来在各数据中心间路由MySQL连接

如果活跃数据中心安装的 MySQL 彻底崩溃了，中间件可以路由流量到另外一个数据中心的服务器池中，应用无须知道这个变化。

这种配置方法的主要缺点是在一个数据中心的 Apache 服务器和另外一个数据中心的 MySQL 服务器之间的延迟比较大。为了缓和这个问题，可以把 Web 服务器设置为重定向模式。这样通信都会被重定向到放置活跃 MySQL 服务器的数据中心。还可以使用 HTTP 代理来实现这一目标。

图 12-1 显示了如何使用代理来连接 MySQL 服务器，也可以将这个方法和许多别的中间件架构结合在一起，例如 LVS 和硬件负载均衡器。

12.5.4 在应用中处理故障转移

有时候让应用来处理故障转移会更简单或者更加灵活。例如，如果应用遇到一个错误，这个错误外部观察者正常情况下是无法察觉的，例如关于数据库损坏的错误日志信息，那么应用可以自己来处理故障转移过程。

虽然把故障转移处理过程整合到应用中看起来比较吸引人，但可能没有想象中那么有效。大多数应用有许多组件，例如 *cron* 任务、配置文件，以及用不同语言编写的脚本。将故障转移整合到应用中可能导致应用变得太过笨拙，尤其是当应用增大并变得更加复杂时。

586 但是将监控构建到应用中是一个好主意，当需要时，能够立刻开始故障转移过程。应用应该也能够管理用户体验，例如提供降级功能，并显示给用户合适的信息。

12.6 总结

可以通过减少宕机来获得高可用性，这需要从以下两个方面来思考：增加两次故障之间的正常运行时间（MTBF），或者减少从故障中恢复的时间（MTTR）。

要增加两次故障之间的正常运行时间，就要尝试去防止故障发生。悲剧的是，在预防故障发生时，它仍然会觉得你做的不够多，所以预防故障的努力经常会被忽视掉。我们已经着重提到了如何在 MySQL 系统中预防宕机；具体的细节可以参阅我们的白皮书，从 *http://www.percona.com* 上可以获得。试着从宕机中获得经验教训，但也要谨防在故障根源分析和事后检验时集中在某一点上而忽略其他因素。

缩短恢复时间可能更复杂并且代价很高。从简单和容易的方面来说，可以通过监控来更快地发现问题，并记录大量的度量值以帮助诊断问题。作为回报，有时候可以在发生宕机前就发现问题。监控并有选择地报警以避免无用的信息，但也要及时记录状态和性能度量值。

另外一个减少恢复时间的策略是为系统建立冗余，并使系统具备故障转移能力，这样当故障发生时，可以在冗余组件间进行切换。不幸的是，冗余会让系统变得相当复杂。现在应用不再是集中化的，而是分布式的，这意味着协调、同步、CAP 定理、拜占庭将军问题，以及所有其他各种杂乱的东西。这也是像 NDB Cluster 这样的系统很难创建并且很难提供足够的通用性来为所有的工作负载提供服务的原因。但这种情况正在改善，也许到本书第四版的时候我们就可以称赞一个或多个集群数据库了。

本章和前面两章提及的话题常常被放在一起讨论：复制、可扩展性，以及高可用性。我们已经尽量将它们独立开来，因为这有助于理清这些话题的不同之处。那么这三章有哪些关联之处呢？

在其应用增长时，人们一般希望从他们的数据库中知道三件事：

- 他们希望能够增加容量来处理新增的负载而不会损失性能。
- 他们希望保证不丢失已提交的事务。
- 他们希望应用能一直在线并处理事务，这样他们就能够一直赚钱。

为了达到这些目的，人们常常首先增加冗余。结合故障转移机制，通过最小化 MTTR 来提供高可用性。这些冗余还提供了空闲容量，可以为更多的负载提供服务。 587

当然，除了必要的资源外，还必须要有一份数据副本。这有助于在损失服务器时避免丢失数据，从而增强持久性。生成数据副本的唯一办法是通过某种方法进行复制。不幸的是，数据副本可能会引入不一致。处理这个问题需要在节点间协调和通信。这给系统带来了额外的负担；这也是系统或多或少存在扩展性问题的原因。

数据副本还需要更多的资源（例如更多的硬盘驱动器，更多的 RAM），这会增加开销。有一个办法可以减少资源消耗和维护一致性的开销，就是为数据分区（分片）并将每个分片分发到特定的系统中。这可以减少需要复制的重复数据的次数，并从资源冗余中分离数据冗余。

所以，尽管一件事总会导致另外一件事，但我们是在讨论一组相关的观点和实践来达成一系列目的。他们不仅仅是讲述同一件事的不同方式。

最后，需要选择一个对你和应用有意义的策略。决定选择一个完全的端到端（end-to-end）高可用性策略并不能通过简单的经验法则来处理，但我们给出的一些粗略的指引也许会有所帮助。

为了获得很短的宕机时间，需要冗余服务器能够及时地接管应用的工作负载。它们必须在线并一直执行查询，而不仅仅是备用，因此它们是“预热”过的，处于随时可用的状态。

如果需要很强的可用性保证，就需要诸如 MySQL Cluster、Percona XtraDB Cluster，或者 Clustrix 这样的集群产品。如果能容忍在故障转移过程中稍微慢一些，标准的 MySQL 复制也是个很好的选择。要谨慎使用自动化故障转移机制；如果没有按照正确的方式工作，它们可能会破坏数据。

如果不是很在意故障转移花费的时间，但希望避免数据丢失，就需要一些强力保证数据的冗余——例如，同步复制。在存储层，这可以通过廉价的 DRBD 来实现，或者使用两个昂贵的 SAN 来进行同步复制。也可以选择在数据库层复制数据，可以使用的技术包括 MySQL Cluster、Percona XtraDB Cluster 或者 Clustrix。也可以使用一些中间件，例如 Tungsten Replicator。如果不需要强有力的保护，并且希望尽量保证简单，那么正常的异步复制或半同步复制在开销合理时可能是很好的选择。

或者也可以将应用放到云中。为什么不呢？这样难道不是能够立刻获得高可用性和无限扩展能力吗？下一章将继续探讨这个问题。

第 13 章
云端的MySQL

许多人在云中使用 MySQL，有时候规模还非常庞大，这并不奇怪。从我们的经验来看，大多数人使用的是 Amazon Web Services 平台（AWS）：特别是 Amazon 的弹性计算云（Elastic Compute Cloud，EC2），弹性块存储（Elastic Block Store，EBS），以及更小众的关系数据库服务（Relational Database Service，RDS）。

为了便于讨论 MySQL 在云中的应用，可以将其粗略分为两类。

IaaS（基础设施即服务）

Iaas 是用于托管自有的 MySQL 服务器的云端基础架构。可以在云端购买虚拟的服务器资源来安装运行 MySQL 实例。也可以根据需求随意配置 MySQL 和操作系统，但没有权限也无法看到处于底层的物理硬件设备。

DBaaS（数据库即服务）

MySQL 本身作为由云端管理的资源。用户需要先收到 MySQL 服务器的访问许可（通常是一个连接串）才能访问。也可以配置一些 MySQL 选项，但没有权限去控制或查看底层的操作系统或虚拟服务器实例。例如 Amazon 运行 MySQL 的 RDS。其中一些服务器并非真的使用 MySQL，但它们能兼容 MySQL 协议和查询语言。

我们讨论的重点主要集中在第一类：云托管平台，例如 AWS、Rackspace Cloud 以及 Joyent[注1]。有许多很好的资源介绍如何部署和管理 MySQL 及其运行所需要的资源，并且也有非常多的平台来完全满足这样的需求，所以我们不会展示代码样例或讨论具体的操作技术。因此，本章关注的重点是，在云端运行 MySQL 还是在传统服务器上部署 MySQL，它们在最终经济上和性能特性上的关键区别是什么。我们假定你对云计算很熟悉。这里不是对云计算概念的简单介绍，我们的目的只是帮助那些还不熟悉在云端部署 MySQL 的用户在使用时避免一些可能遇到的陷阱。

注 1： OK，我们承认。Amazon 网络服务是一个云。本章主要讨论 AWS。

590 一般来说，MySQL 能够在云中很好地运行。在云中运行 MySQL 并不比在其他平台困难，但有一些非常重要的差别。你需要注意这些差别并据此设计应用和架构来获得好的效果。某些场景下在云端托管 MySQL 并不是非常适合，有时候则很适合，但大多数时候云仅仅是另外一个部署平台而已。

云是一个部署平台，而不是一种架构，理解这一点很重要。架构会受平台的影响，但平台和架构明显不同。如果你把架构和平台搞混了，就可能会做出不合适的选择而给以后带来麻烦。这也正是我们要花时间讨论云端的 MySQL 到底有什么不同的原因。

13.1 云的优点、缺点和相关误解

云计算有许多优点，但很少是为 MySQL 特别设计。有一些书籍已经介绍了相关的话题[注2]，这里我们不再赘述。不过我们会列出一些比较重要的条目供参考，因为接下来会讨论到云计算的缺点，我们不希望你认为我们是在过分苛求云计算。

- 云是一种将基础设施外包出去无须自己管理的方法。你不需要寻找供应商购买硬件，也不需要维护和供应商之间的关系，更无须替换失效的硬盘驱动器等。
- 云一般是按照即用即付的方式支付，可以把前期的大量资本支出转换为持续的运营成本。
- 随着供应商发布新的服务和成本降低，云提供的价值越来越大。你自己无须做任何事情（例如升级服务器），就可以从这些提升中获益；随着时间推移你会很容易地获得更多更好的选择并且费用更低。
- 云能够帮助你轻松地准备好服务器和其他资源，在用完后直接将其关闭，而无须关注怎么处理它们，或者怎么卖掉它们收回成本。
- 云代表了对基础设施的另一种思考方式——作为通过 API 来定义和控制的资源——支持更多的自动化操作。从“私有云”中也可以获得这些好处。

当然，不是所有跟云相关的东西都是好的。这里有一些缺点可能会构成挑战（在本章稍后部分我们会列出 MySQL 特有的缺点）。

- 资源是共享并且不可预测的，实际上你可以获得比你支付的更多的资源。这听起来
  591 很不错，但却导致容量规划很难做。如果你在不知情的情况下获得了比理应享受到的更多的计算资源，那么就存在这样的风险：别人也许会索要他们应得的资源，这会使你的应用性能退化到应有的水平。一般来说，很难确切地知道本来应该得到多少（资源），大多数云托管服务提供商不会对此给出确切的答案。
- 无法保证容量和可用性。你可能以为还可以获得新实例，但如果供应商已经超额销

注 2：参阅 George Reese 所写的 *Cloud Application Architectures*（O'Reilly）。

售了呢？这在有很多共享资源的情况下会发生，同样也会发生在云中。

- 虚拟的共享资源导致排查故障更加困难，特别是在无法访问底层物理硬件的情况下无法检查并弄清到底发生了什么。例如，我们曾经看到过一些系统的 *iostat* 显示的 I/O 很正常或者 *vmstat* 显示的 CPU 很正常，而当实际衡量完成一个任务需要的时间时，资源却被系统上的其他东西严重占用了。如果在云平台上出现了性能问题，尤其需要去仔细地分析检测。如果对此并不擅长，可能就无法确认到底是底层系统性能差，还是你做了什么事情导致应用出现不合理的资源需求。

总的来说，云平台上对性能、可用性和容量的透明性和控制力都有所下降。最后，还有一些对云的误解需要记住。

云天生具备更好的可扩展性

应用、云的架构，以及管理云服务的组织是不是都是可扩展的。云并不是天生可扩展的，云也仅仅是云而已，选择一个可扩展的平台并不能自动使应用变得可扩展。的确，如果云托管提供商没有超售，那么你可以根据需求来购买资源，但在需要时能够获得资源仅仅是扩展性的一个方面而已。

云可以自动改善甚至保证可用时间

一般来说，个别在云端托管的服务器比那些经过良好设计的专用基础设施更容易发生故障或运行中断。但是许多人并没有意识到这一点。例如，有人这样写道："我们将基础设施升级到基于云构建的系统以保证 100% 的可用时间和可扩展性"。而就在这之前 AWS 遭受了两次大规模的运行中断故障，导致很大一部分用户受影响。好的架构能够用不可靠的组件设计出可靠的系统，但通常更可靠的基础设施可以获得更高的可用性。（当然不可能有 100% 的可用时间的系统。）

另一方面，购买云计算服务，实际上是购买一个由专家构建的平台。他们已经考虑了许多底层的东西，这意味着你可以更专注于上层工作。如果构建自己的平台而对其中的那些细枝末节并不精通，就可能犯一些初学者的错误，早晚会导致一些宕机时间。从这一点来说，云计算能够帮助改善可用时间。 592

云是唯一能提供 [这里填入任意的优点] 的东西

事实上，许多云的优点是继承自构建云平台所用到的技术，即使不使用云也可以获得[注3]。例如，通过管理得当的虚拟化和容量规划，可以像任何一个云平台那样简单快速地启动（spin up）一台新的机器。完全没必要专门使用云来做到这一点。

云是一个"银弹"（silver bullet）

虽然大部分人会认为这很荒谬，但确实有人会这么认为。实际上完全没有这回事。

无可否认，云计算提供了独特的优点，随着时间的推移，关于云计算是什么，以及它们

注 3： 我们不是说这会更加容易或便宜，我们只是说云并不是能获得这些好处的唯一途径。

在什么情况下会有帮助，我们会获得更多的共识。但有一点非常肯定：它是全新的，我们现在所做的任何预测都未必经得起时间的考验。我们会在本书讨论相对安全的部分，而将剩下的部分留给读者讨论。

13.2 MySQL 在云端的经济价值

在一些场景下云托管比传统的服务器部署方式更经济。以我们的经验来看，云托管比较适合尚处于初级阶段的企业，或者那些持续接触新概念并且本质上是以适用为主的企业，例如移动应用开发者或游戏开发者。这些技术的市场随着移动计算的扩张出现了爆炸式增长，并且仍然是快速发展的领域。在许多情况下，成功的因素并不为开发者所控制，例如口口相传的推荐或者恰逢重要国际事件的时机。

我们已经帮助很多公司在云中构建移动应用、社交网络以及游戏应用。其中一个他们大量使用的策略是尽可能又快又便宜地开发和发布应用。如果一个应用碰巧变得流行了，公司将投入资源扩大其规模；否则就会很快终结这些应用。一些公司构建并发布的应用的生命周期甚至只有几个星期，在这样的环境下，可以毫不犹豫地选择云托管。

如果是一个小规模的公司，可能无法提供足够的硬件来自建数据中心以满足一个非常流行的 Facebook 应用的发展曲线。我们也协助过一些大型的 Facebook 应用进行扩展，它
593 们能够以令人惊讶的速度增长——有时甚至会快到让一个主机托管公司耗尽资源。更为
严重的是，这些应用的增长是完全无法预测的；它们可能只有极少量的用户（也可能突然有了爆炸性的用户数量增长）。我们在数据中心和云中都遇到过这样的应用。如果是一个小公司，云可以帮你避免前期快速注入大量的资金来获得更快更大规模的风险。

云的另一种潜在的大用途是运行不是很重要的基础设施，例如集成环境、开发测试平台，以及评估环境。假设部署周期是两个星期。你会每天每个小时都测试部署一次，还是只在项目最后的冲刺时测试？许多用户只是偶尔需要筹划和部署测试环境。在这种场景下，云可以帮助节约不少钱。

以下是我们使用云的两种方式。第一个是作为我们对技术职员面试的一部分，我们会询问如何解决一些实际的问题。我们使用 AMI（Amazon Machine Images）来模拟一些被“破坏”的机器，然后让求职者登录并在服务器上执行一系列任务。我们不必开放他们到内部网络的授权，这种方案显然要方便得多。另一个是作为新项目的工作平台和开发服务器。有一个这样的项目已经在一台云端开发服务器上运行了数个月，而花费不足一美元！这在我们自己的基础设施上是不可能做到的。单是发送一封邮件给系统管理员申请开发服务器的时间价值就不止一美元。

但是另一方面，云托管对于长期项目而言可能会更加昂贵。如果打算长远地使用云，就需要花时间来计算一下（它是否划算）。除了猜想未来的创新能给云计算和商用硬件带来什么，还需要做基准测试以及一个完整的总体持有成本（TCO）账单。为了理清事情的本质并考虑全面所有相关的细节，你需要把所有的事情最终归结为一个数字：每美元的业务交易数。事情变化得太快，所以我们将这个留给读者思考。

13.3 云中的 MySQL 的可扩展性和高可用性

正如我们之前提到的，MySQL 并不会在云端自动变得更具扩展性。事实上，如果机器的性能较差，会导致过早使用横向扩展策略。况且云托管服务器相比专用的硬件可靠性和可预测性要更差些，所以想在云端获得高可用性需要更多的创新。

但是总的来说，在云端中扩展 MySQL 和在其他地方扩展没有太多的差别。最大的不同就是按需提供服务器的能力。但是也有某些限制会导致扩展和高可用实现起来有点麻烦，至少在有些云环境中是这样的。例如，在 AWS 云平台中，无法使用类似虚拟 IP 地址的 594
功能来完成快速原子故障转移。像这种对资源的有限控制意味着你需要使用其他办法，例如代理。（ScaleBase 也值得去看看。）

云另外一个迷惑人的地方是梦想中的自动扩展——就是根据需求的增加或减少来启动或关闭实例。尽管对于诸如 Web 服务器这样的无状态部分是可行的，但对于数据库服务器而言则很难做到，因为它是有状态的。对于一些特定的场景，例如以读为主的应用，可以通过增加备库的方式来获得有限的自动扩展[注4]，但这并不是一个通用的解决方案。实际上，虽然许多应用在 Web 层使用了自动扩展，但 MySQL 并不具备在一个无共享（Shared Nothing）集群中的对等角色服务器之间迁移的能力。你可以通过分片架构来自动重新分片并自动增长或收缩[注5]，但 MySQL 本身是无法自动扩展的。

事实上，因为数据库通常是一个应用系统中主要或唯一的有状态并且持久化的组件，所以把应用服务迁移到云端是很普遍的事情，因为除数据库之外的所有部分都可以从云中收益——Web 服务器、工作队列服务器、缓存等——而 MySQL 只需要处理剩下的东西。

毕竟，数据库并非世界的中心。如果应用系统其他部分获得的好处，超过了让 MySQL 运行得足够好而投入的额外开销和必需的工作量，那这不是一个是否会发生的问题，而是怎么发生的问题。要回答这个问题，最好先了解你在云中可能碰到的额外的挑战。这些通常围绕着数据库服务器的可用资源。

注 4： Scalr (*http://scalr.net*) 是一个流行的开源服务，用于在云中进行 MySQL 复制自动扩展。
注 5： 计算机科学家喜欢将之称为“重大挑战”（non-trivial challenge）。

13.4 四种基础资源

MySQL 需要四种基础资源来完成工作：CPU 周期、内存、I/O，以及网络。这四种资源的特性和重要程度在不同的云平台上各不相同。可以通过了解它们的不同之处和对 MySQL 的影响，以决定是否选择在云中托管 MySQL。

- CPU 通常很少且慢。在写作本书时最大的标准 EC2 实例提供 8 个虚拟 CPU 核心。EC2 提供的虚拟 CPU 比高端 CPU 的速度明显要慢很多（可以查看本章稍后的基准测试结果）。虽然可能略有不同，但很可能在大多数云托管平台中这都是一种普遍现象。EC2 提供使用多个 CPU 资源的实例，但它们的最大可用内存却更低。在写作本书时商用服务器能提供几十个 CPU 核心——甚至更多，如果按硬件线程算的话。[注6]
595
- 内存大小受限制。最大的 EC2 实例当前能提供 68.4GB 的内存。与此相比，商用服务器能提供 512GB ～ 1TB 的内存。
- I/O 的吞吐量、延迟以及一致性受到限制。在 AWS 云中有两个存储选项。

  第一个选择是使用 EBS 卷，这有点类似云中的 SAN。AWS 的最佳实践是在用 EBS 组建的 RAID10 卷上建立服务器。但是 EBS 是一个共享资源，就像 EC2 服务器和 EBS 服务器之间的网络连接。延迟可能会很高并且不可预测，即使是在适量的吞吐量需求下也是如此。我们已经测得 EBS 设备的 I/O 延迟可以达到十几分之一秒。相比之下，直接插在本机的商用硬盘驱动器只需几个毫秒，而闪存设备比硬盘驱动器的速度又要高出几个数量级。但另一方面，EBS 卷也有许多很好的特性，例如和其他 AWS 服务、快照等结合起来使用。

  第二个选择是实例的本地存储。每个 EC2 服务器有一定数量的本地存储，实际安装在底层服务器上。它能够比 EBS 提供更多的一致性性能[注7]，但如果实例停止了就无法做到持久化。正是由于这样的特性导致其不适合大多数的数据库服务器场景。
- 尽管网络通常是一个变化多端的共享资源，但是性能通常比较好。虽然使用商用硬件可以获得更快更持续的网络性能，但 CPU、RAM 和 I/O 更容易成为主要的性能瓶颈，在 AWS 云中我们还没有遇到过网络性能问题。

正如你所看到的，四种基础资源中有三种在 AWS 云中是受限的，在某些场景下尤其明显。总的来说，这些基础资源并没有商业硬件那样的性能。下一节我们会讨论这些确切的结论。

注 6： 在 CPU、RAM 以及 I/O 上，商用硬件能够提供超过 MySQL 可以有效利用的硬件能力，所以将云与云之外可获得的最强硬件相比较并不是完全公平的。

注 7： 直到写入的时候本地存储才会被分配给实例，导致每个写入的块发生“第一次写处罚”（first-write penalty）。避免这个问题的办法是使用 *dd* 去写满设备。

13.5 MySQL 在云主机上的性能

通常，由于较差的 CPU、内存以及 I/O 性能，在类似 AWS 这样的云托管平台上 MySQL 所表现出来的性能并不如在其他地方好。这些情况在不同的云平台之间略有不同，但这依然是普遍的事实[注8]。然而对于你的需求而言，云主机可能仍然是一个性能足够高的平台，在某些需求上云平台可能比另外的解决方案要好。

如果使用更糟糕的硬件来运行 MySQL，无法让 MySQL 性能比托管在云平台上更高，这并不奇怪。真正让人感到困惑的是在相似规格的物理硬件条件下却无法获得同样的运行速度。例如，如果有一台服务器拥有 8 个 CPU 核心，16GB 内存以及一个中等的 RAID 阵列，你可能认为能够获得和一个拥有 8 个 EC2 计算单元、15GB 内存以及少量 EBS 卷的 EC2 实例相同的性能，但这是无法保证的。EC2 实例的性能可能比你的物理硬件更加多变，特别是它不是一个超大实例时，可以推测它跟其他实例共享了同样的硬件资源。 596

稳定性确实非常重要。MySQL 和 InnoDB 尤其不喜欢不稳定的性能——特别是不稳定的 I/O 性能。I/O 操作会请求服务器内部的互斥锁，当持续时间太长时，就会显著地导致很多"阻塞"进程堆积起来，出现令人难以理解的长时间运行的查询语句，以及例如 `Threads_running` 或 `Threads_connected` 这样的状态变量产生毛刺。

实际应用中前后不一致或者无法预测的性能导致的结果就是排队变得越来越严重。排队是响应时间和到达间隔时间多变自然会导致的结果，并且有个完整的数学分支专门致力于排队的研究。所有的计算机都是队列系统的网络，当需要请求的资源（CPU、I/O，网络，等等）繁忙时，请求必须等待。当资源性能更加多变时，请求更容易堆叠，会出现更多的排队现象。因此，在大多数云计算平台上很难获得高并发或者稳定的低响应时间。我们有很多次在 EC2 平台上遭受到这个限制的经验。以我们的经验来看，即便在最大的实例上运行的 MySQL，在典型的 Web OLTP 工作负载上，你能够期待的最高并发度也就是 `Threads_running` 值为 8 ~ 12。根据经验，当超过这个值时，性能会越来越不可接受。

注意我们所说的"典型的 Web OLTP 工作负载"，并非所有的工作负载都以相同的方式反映云平台的限制。确实有一些工作负载在云中表现得很好，而有一些则受到严重影响，让我们看看到底有哪些。

- 正如我们刚讨论的，需要高并发的工作负载并不是非常适合云计算。对于那些要求非常快的响应时间的应用同样如此。原因可以归结于虚拟 CPU 的数目和速度方面的限制。每个 MySQL 查询运行在一个单独的 CPU 上，所以查询响应时间实际上是由 CPU 的原始速度决定的。如果期望得到更快的响应时间，就需要更快的 CPU。为了

注 8： 如果你相信 *http://www.xkcd.com/908/*，那么显然所有的云都有同样的缺点，我们刚刚已经提过。

支持更高的并发度，你需要更多的 CPU。MySQL 和 InnoDB 不会因为运行在大量 CPU 核心上而提供爆炸式的改进，但目前通常能在至少 24 个核心上获得比较好的横向扩展，这通常比在云中能够获得的核心数更多。

- 那些需要大量 I/O 的工作负载在云中并不总是表现很好。当 I/O 很慢并且不稳定时，工作会很快中断。但另一方面，如果你的工作负载不需要太多的 I/O，不管是吞吐量(每秒的执行量）还是带宽（每秒字节数），MySQL 就可以运行得很好。

597 之前的几点是根据云端的 CPU 和 I/O 资源的缺点得出的。那么关于这些你可以做点什么呢？对于 CPU 限制你做不了太多，不够就是不够。但是 I/O 则不同。I/O 实际上是两种存储器的交换：非永久存储器（RAM）和持久化存储器（磁盘、EBS，或者其他你所拥有的)。因此 MySQL 的 I/O 需求会受系统内存大小的影响。当有足够的内存时，可以从缓存中读取数据，从而减少读和写操作的 I/O。写入同样可以缓存在内存里，多个对相同内存比特位的写入可以合并成单个 I/O 操作。

内存的限制就出现了。当拥有足够的内存来存放工作数据集时[注9]，某些工作负载的 I/O 需求可以明显减少。更大的 EC2 实例也会提供更好的网络性能，更有利于 EBS 卷的 I/O。但如果工作集太大，无法装入可用的最大实例，则 I/O 需求会逐渐上升，并开始阻塞甚至停止服务，正如我们之前讨论的那样。EC2 中内存最大的实例能够很好地为许多工作负载提供足够的内存。但是你需要意识到，预热时间可能会很长；关于这一话题本节后面会有更多的讨论。

哪种类型的工作负载无法通过增加更多的内存来解决呢？除了缓存外，一些写入很大的工作负载需要的 I/O 比你能从多数云计算平台上获得的要多。例如，如果每秒执行事务数很多，那么每秒就需要执行更多的 I/O 操作以保证持久性。你只能从诸如 EBS 这样的系统中获得这么多的吞吐量。同样地，如果你正在将大量数据写入到数据库中，可能会超过可用的带宽。

你可能认为通过 RAID 来为 EBS 卷进行条带（striping）和镜像可以改善 I/O 性能。在某种程度上确实有帮助。问题是，当增加更多的 EBS 卷时，在我们需要某个 EBS 卷的任意时间点都增加了它性能变差的可能性，而根据 InnoDB 内部 I/O 工作的方式，最差的一环通常是整个系统的瓶颈。实际上，我们已经尝试过 10 和 20 个 EBS 卷的 RAID 10 集合，20 卷的 RAID 比 10 卷的遭遇了更多的停顿（stall）问题。当我们测量底层块设备的 I/O 性能时，很明显只有一或两个 EBS 卷表现得很慢，但是却已经影响了整个系统。

你也可以改变应用和服务器来减少 I/O 需求，考虑周到的逻辑和物理数据库设计（Schema 和索引）对于减少 I/O 请求大有帮助，应用程序优化和查询优化也一样。这是减少 I/O

注 9： 参阅第 9 章关于工作集的定义及其如何影响 I/O 需求的讨论。

最有效的手段。例如插入量很大的工作负载，明智地使用分区，将 I/O 集中到索引能完全加载到内存中的单个分区上，就会有所帮助。你也可以通过设置 `innodb_flush_logs_at_trx_commit=2` 和 `sync_binlog=0` 来降低持久性，或者将 InnoDB 事务日志和二进制 598
日志从 EBS 卷中转移到一个本地驱动器上（尽管这有风险）。但是你从服务器上压榨一点额外的性能越困难，就越不可避免地要引入更大的复杂性（以及它们的成本）。

此外还可以升级 MySQL 服务器软件。新版本的 MySQL 和 InnoDB（最新的使用 InnoDB Plugin 的 MySQL 5.1，或者 MySQL 5.5 及更新的版本）能够提供更好的 I/O 性能以及更少的内部瓶颈，并且相比 5.1 及之前的版本遭受的停顿和堆积会少很多。Percona Server 在某些工作负载下能够提供更多的好处。例如，Percona Server 的快速预热缓冲池特性在服务器重启后能够帮助备用服务器快速运行起来，特别是 I/O 性能不是很好并且服务器依赖于内存时。这也是我们讨论能在云中获得好的性能的候选场景，这里服务器比备用硬件更容易发生故障。Percona Server 能够将预热时间从几个小时甚至几天减少到几分钟。在写作本书时，类似的预热特性在 MySQL 5.6 的开发里程碑版本里已经可用了。

尽管最终一个增长的应用总会达到一个顶点，届时你不得不对数据库进行拆分以保证数据能够存放到云中。我们倾向于尽量不拆分，但如果你只有这么点马力，当达到某个点时，就不得不去其他地方（离开这个云），或者将其拆分为多份，使每份数据需要的资源不超过虚拟硬件能提供的。通常当工作集无法适应内存大小时就得要进行分片了，这意味着在最大的 EC2 实例上的工作集大小为 50GB ~ 60GB。与之相对，我们已经有很多在物理硬件上运行几个 TB 大小级别数据库的经验。在云中你需要更早进行分片。

13.5.1 在云端的 MySQL 基准测试

我们进行了一些基准测试以说明 MySQL 在 AWS 云环境中的性能。当需要大量 I/O 时要在云中获得始终稳定并且可重现的基准测试结果几乎是不可能的，所以我们选择一个内存中的工作负载，本质上可以衡量除了 I/O 外的所有因素。我们使用 Percona Server 5.5.16，缓冲池为 4GB，在一千万行数据上运行标准 SysBench 只读基准测试。这样就可以根据不同的实例大小进行比较。我们忽略了高频率 CPU 实例，因为它们实际上比 *m2.4xlarge* 实例的 CPU 性能要差。我们还引用了一台 Cisco 服务器作为参考。Cisco 机器性能非常高但有点老化了，使用的是两个 2.93GHz 的 Xeon X5670 Nehalem CPU。每个 CPU 有 6 个核心，每个核心上有两个硬件线程，在操作系统来看总共有 24 个 CPU。图 13-1 显示了测试的结果。

599

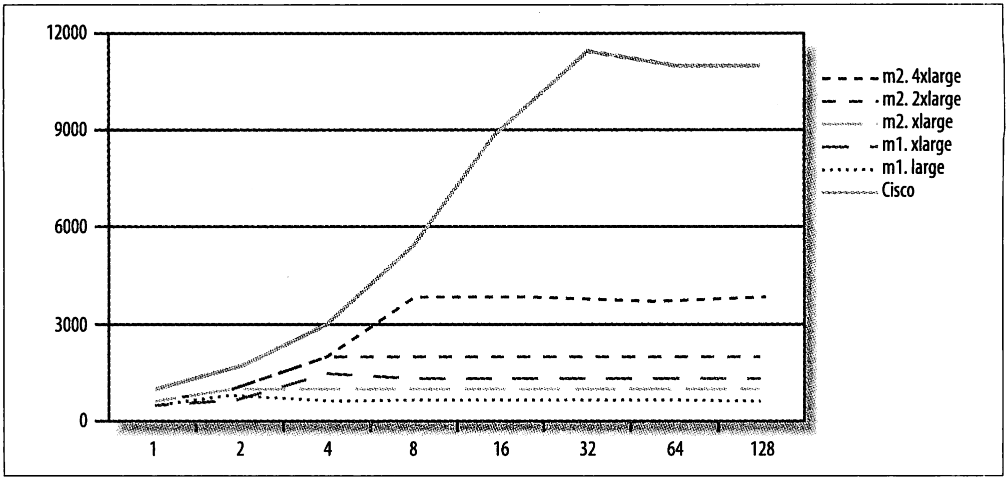


图13-1：使用SysBench对AWS云中的MySQL进行只读基准测试

根据工作负载和硬件来看，这样的结果并不奇怪。例如，最大的 EC2 实例最高有 8 个线程，因为它有 8 个 CPU 核心。（读 / 写工作负载会花费一些 CPU 之外的时间来做 I/O，所以我们能获得超过 8 个线程的有效并发度）。图 13-1 可能会让你认为 Cisco 的优势就是 CPU 能力，这也是我们原本认为的。所以我们使用 SysBench 的质数基准测试来测试原始 CPU 性能。结果如图 13-2 所示。

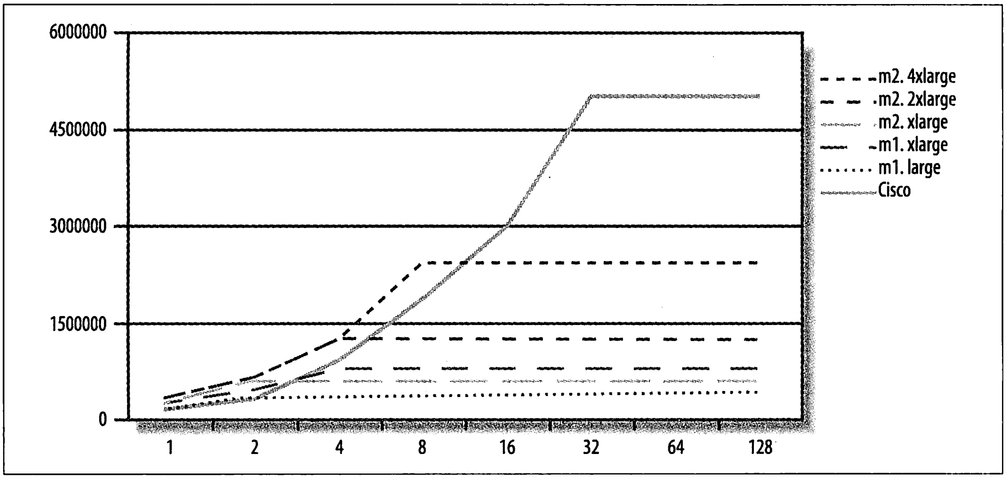


图13-2：使用SysBench对AWS服务器进行CPU质数基准测试

Cisco 服务器每个 CPU 的性能比 EC2 服务器要低，奇怪么？我们也感到非常奇怪。质数基准测试本质上是原始 CPU 指令，因此不应该有非常明显的虚拟化开销或者太多的内

存交换。对于这样的结果我们的解释是这样的：Cisco 服务器的 CPU 已经使用了很多年了，并且比 EC2 服务器的要慢。但是对于一些更加复杂的任务，例如运行数据库服务器，EC2 服务器会受到虚拟化开销的影响。区分慢 CPU、慢内存访问以及虚拟化开销并不总是很容易，但在这个实例中这种区别看起来很明显。 600

13.6 MySQL 数据库即服务（DBaaS）

在云端服务器上安装 MySQL 并不是在云中使用 MySQL 的唯一方法。已经有越来越多的公司开始将数据库本身作为云资源，称之为数据库即服务（DBaaS，有时候也叫 DaaS），这意味着你可以在一个地方使用云中的数据库，而在另外的地方运行真正的服务。虽然我们在本章花很多时间解释了 IaaS，但 IaaS 市场正在快速商品化，我们期望未来重点会转到 DBaaS。在写作本书时已经有以下几个 DBaaS 服务提供商。

13.6.1 Amazon RDS

我们发现在 Amazon 的关系数据库（RDS）上进行的开发比其他任何一个 DBaaS 提供商都要多很多。Amazon RDS 不仅仅是一个兼容 MySQL 的服务；它事实上就是 MySQL，所以能够完全兼容你所拥有的 MySQL 服务器[注10]并能作为替代品提供服务。我们不是很确定，但如大多数人一样，我们相信 RDS 是托管在使用 EBS 卷的 EC2 机器上——Amazon 并没有公布底层的技术，但当你足够了解 RDS 时，这看起来很明显就是 MySQL、EC2 以及 EBS。

系统管理职责完全由 Amazon 来承担。你没有访问 EC2 机器的权限；只有登入 MySQL 的访问凭证。你可以创建数据库、插入数据等。你并没有被控制住，如果有需要，可以将数据导出来转移到其他地方，也可以创建卷快照并挂载到其他机器上。

为了防止你检查或干涉 Amazon 对服务器或主机实例的管理，RDS 做了一些限制。例如一些权限限制。你不能利用 `SELECT INTO OUTFILE`、`FILE()`、`LOAD DATA INFILE` 或其他 601
方法来通过 MySQL 访问服务器的文件系统。你不能做任何和复制相关的事情，也不能为自己赋予更高的权限。Amazon 通过诸如在系统表上设置触发器等方法来进行阻止。并且作为服务条款的一部分，你要同意不会试图绕过这些限制。

安装的 MySQL 版本做了轻微的修改以阻止用户干涉服务器，其他部分看起来和原版 MySQL 一样。我们对 RDS、EBS 和 EC2 做了基准测试，并没有从该平台上发现超出我们预期的变化。也就是说，看起来 Amazon 并没有对服务器做任何性能增强。

RDS 可以提供一些比较吸引人的好处，这取决于你的具体情况。

注 10：除非你使用别的存储引擎或者其他一些非标准的 MySQL 修改版本。

- 你可以将系统管理甚至许多数据库管理的工作留给 Amazon。例如，他们会为你进行复制并保证你不会把事情搞砸。
- RDS 相比其他选择而言可能更便宜，这取决于你的成本结构和人力资源。
- RDS 中的限制也许是件好事：Amazon 拿走了那把子弹上膛的枪，防止你用它自残。

但是，它也有一些潜在的缺点。

- 由于无法控制服务器，也就无法弄清操作系统中到底发生了什么。例如，你无法衡量 I/O 响应时间和 CPU 利用率。Amazon 通过另一个服务 CloudWatch 提供了这一功能。它给出了足够的指标用于排查许多性能问题，但有时候你需要原始数据以知道到底发生了什么。（也无法使用类似 `FILE()` 这样的函数来访问 `/proc/diskstats`。）
- 无法获得完整的慢查询日志文件。你可以指定 MySQL 将慢查询记录到一个 CSV 日志表中，但这并不是很好。它会消耗很多服务器资源，并且不会给出精确的查询响应时间。这使得很难去分析和排除 SQL 故障。
- 如果你希望得到最新最好的，或者一些性能上的增强，例如那些你可以从 Percona Server 上获得的提升，那就不走运了，RDS 并不提供这些。
- 你必须依赖 Amazon 的支持团队来解决一些问题，而这些问题可能本来是你自己可以解决的。例如，假设查询挂起了，或者服务器由于数据损坏崩溃了。你既可以等待 Amazon 来解决，也可以自己解决。如果是后者你就需要把数据转移到别的地方。你无法通过访问实例本身来解决。如果想这么做，你不得不额外花一些时间并支付额外的资源。这不只是理论上的推测；我们已经接到过许多技术支持请求，这些请求通常需要系统权限以进行故障排查，因此对于 RDS 用户而言是无法真正解决的。

602 正如我们所说，在性能方面，RDS 跟一个大型大内存的使用 EBS 存储和原始 MySQL 的 EC2 实例相似。如果直接使用 EC2 和 EBS 并安装一个高性能版本的 MySQL（例如 Percona Server），你可以从 AWS 云中压榨出一点更高的性能，但这不会是一个数量级上的区别。考虑到这一点，有理由根据你的商业需求而非性能需求来决定是否使用 RDS。如果确实非常要求高性能，那你根本就不应该使用 AWS 云。

13.6.2 其他 DBaaS 解决方案

Amazon RDS 并不是 MySQL 用户唯一可选的 DBaaS 解决方案。还有诸如 FathomDB（*http://fathomdb.com*）以及 Xeround（*http://xeround.com*）等服务。但我们并没有足够的第一手经验来介绍它们，因为我们还没有在这些服务上做任何的生产部署。从关于 FathomDB 的一些有限的公开信息来看，它和 Amazon RDS 有点类似，虽然它也和 AWS 云一样可以在 Rackspace 云上获得。在写作本书时它还处于内部测试阶段。

Xeround 则有很大的不同之处：它是一个分布式服务器集群，前端是一个包含特定存储

引擎的 MySQL。它似乎和原始版本 MySQL 有少量的不兼容或不同之处。但它只是最近才发布正式 GA 版本（GA，generally available），所以现在下定论为时尚早。存储引擎似乎是用于和后台集群系统通信，这看起来有点和 NDB CLuster 类似。它增加了自动重分布功能，可以在工作负载增加或减少时自动地增加和去除节点（动态扩展）。

还有许多其他的 DBaaS 服务，新的服务也在不断地推出。我们这里所写的任何内容都可能在你阅读时已经过时了，所以我们将其留给你自己来研究。

13.7 总结

在云端使用 MySQL 至少有两种主流的方法：在云服务器上安装 MySQL，或者使用 DBaaS 服务。MySQL 能够在云主机上运行得很好，但云环境中的限制常常会导致更早需要进行数据拆分。并且尽管云服务器看起来和你的物理硬件很相似，但可能性能和服务质量要更低。

有时候似乎有人会说“云就是答案，有什么问题吗？”这是一个极端，但那些认为云是一个银弹的狂热信众，也有类似的问题。数据库所需要的四种基础资源中的三种（CPU、内存和磁盘）在云中明显更差并且 / 或者效率更低，会直接影响到 MySQL 的性能。

但是对于很多工作负载而言，MySQL 能够在云中运行得很好。通常来说，如果能将工 603
作集加载到内存中，并且产生的写入负载不超过云能支撑的 I/O 量，那么就可以获得很好的效果。通过严谨的设计和架构，选择正确的 MySQL 版本并做合适的配置，可以使你的数据库工作负载和容量能适应云的长处。但是 MySQL 并不是天生的云数据库；也就是说，它无法完全使用云计算理论上能提供的优点，例如自动扩展。但是一些可替代的技术（例如 Xeround）正在尝试解决这些缺点。

我们已经讨论了很多跟云相关的缺点，这也许会给你一个我们反对云计算的印象。并非如此。这只是因为我们只集中在 MySQL 上，而不是讨论云计算所有的优点，这可能跟你从其他地方阅读到的非常不一样。我们在试着指出在云端运行 MySQL 有哪些不同，以及哪些是你需要知道的。

我们看到在云中最大的成功是由于商业原因做出的决策。即使长期来看每个商业交易的开销在云中会更高，但其他方面的因素，诸如增加了弹性、减少了前期成本、减少了推向市场的时间，以及降低了风险，这可能更重要。并且你的应用中其他和 MySQL 无关的部分所获得的好处要远远大于（在云端）使用 MySQL 带来的弊端。

第 14 章 605

应用层优化

如果在提高 MySQL 的性能上花费太多时间，容易使视野局限于 MySQL 本身，而忽略了用户体验。回过头来看，也许可以意识到，或许 MySQL 已经足够优化，对于用户看到的响应时间而言，其所占的比重已经非常之小，此时应该关注下其他部分了。这是个很不错的观点，尤其是对 DBA 而言，这是很值得去做的正确的事。但如果不是 MySQL，那又是什么导致了问题呢？使用第 3 章提到的技术，通过测量可以快速而准确地给出答案。如果能顺着应用的逻辑过程从头到尾来剖析，那么找到问题的源头一般来说并不困难。有时，尽管问题在 MySQL 上，也很容易在系统的另一部分得到解决。

无论问题出在哪里，都至少可以找到一个靠谱的工具来帮助进行分析，而且通常是免费的。例如，如果有 JavaScript 或者页面渲染的问题，可以使用包括 Firefox 浏览器的 Firebug 插件在内的调优工具，或者使用 Yahoo! 的 YSlow 工具。我们在第 3 章提到了几个应用层工具。一些工具甚至可以剖析整个堆栈：New Relic 是一个很好的例子，它可以剖析 Web 应用的前端、应用以及后端。

14.1 常见问题

我们在应用中反复看到一些相同的问题，经常是因为人们使用了缺乏设计的现成系统或者简单开发的流行框架。虽然有时候可以通过这些框架更快更简单地构建系统，但是如果不清楚这些框架背后做了什么操作，反而会增加系统的风险。

下面是我们经常会碰到的问题清单，通过这些过程可以激发你的思维。

- 什么东西在消耗系统中每台主机的 CPU、磁盘、网络，以及内存资源？这些值是否
合理？如果不合理，对应用程序做基本的检查，看什么占用了资源。配置文件通常 606
是解决问题最简单的方式。例如，如果 Apache 因为创建 1000 个需要 50MB 内存的

工作进程而导致内存溢出，就可以配置应用程序少使用一些 Apache 工作进程。也可以配置每个进程少使用一些内存。

- 应用真的需要所有获取到的数据吗？获取 1000 行数据但只显示 10 行，而丢弃剩下的 990 行，这是常见的错误。（如果应用程序缓存了另外的 990 行备用，这也许是有意的优化。）
- 应用在处理本应由数据库处理的事情吗，或者反过来？这里有两个例子，从表中获取所有的行在应用中进行统计计数，或者在数据库中执行复杂的字符串操作。数据库擅长统计计数，而应用擅长正则表达式。要善于使用正确的工具来完成任务。
- 应用执行了太多的查询？ORM 宣称的把程序员从写 SQL 中解放出来的语句接口通常是罪魁祸首。数据库服务器为从多个表匹配数据做了很多优化，因此应用程序完全可以删掉多余的嵌套循环，而使用数据库的关联来代替。
- 应用执行的查询太少了？好吧，上面只说了执行太多 SQL 可能成为问题。但是，有时候让应用来做"手工关联"以及类似的操作也可能是个好主意。因为它们允许更细的粒度控制和更有效的使用缓存，以及更少的锁争用，甚至有时应用代码里模拟的哈希关联会更快（MySQL 的嵌套循环的关联方法并不总是高效的）。
- 应用创建了没必要的 MySQL 连接吗？如果可以从缓存中获得数据，就不要再连接数据库。
- 应用对一个 MySQL 实例创建连接的次数太多了吗（也许因为应用的不同部分打开了它们自己的连接）？通常来说更好的办法是重用相同的连接。
- 应用做了太多的"垃圾"查询？一个常见的例子是发送查询前先发送一个 ping 命令看数据库是否存活，或者每次执行 SQL 前选择需要的数据库。总是连接到一个特定的数据库并使用完整的表名也许是更好的方法。（这也使得从日志或者通过 SHOW PROCESSLIST 看 SQL 更容易了，因为执行日志中的 SQL 语句的时候不用再切换到特定的数据库，数据库名已经包含在 SQL 语句中了。）"预备（Preparing）"连接是另一个常见问题。Java 驱动在预备期间会做大量的操作，其中大部分可以禁用。另一个常见的垃圾查询是 SET NAMES UTF8，这是一个错误的方法（它不会改变客户端库的字符集，只会影响服务器的设置）。如果应用在大部分情况使用特定的字符集工作，可以修改配置文件把特定字符集设为默认值，而不需要在每次执行时去做修改。
- 607 应用使用了连接池吗？这既可能是好事，也可能是坏事。连接池可以帮助限制总的连接数，有大量 SQL 执行的时候效果不错（Ajax 应用是一个典型的例子）。然而，连接池也可能有一些副作用，比如说应用的事务、临时表、连接相关的配置项，以及用户自定义变量之间相互干扰等。
- 应用是否使用长连接？这可能导致太多连接。通常来说长连接不是个好主意，除非网络环境很慢导致创建连接的开销很大，或者连接只被一或两个很快的 SQL 使用，或者连接频率很高导致客户端本地端口不够用。如果 MySQL 的配置正确，也

许就不需要长连接了。比如使用 `skip-name-resolve` 来避免 DNS 反向查询，确保 `thread_cache` 足够大，并且增加 `back_log`。可以参考第 8 章和第 9 章得到更多的细节。

- 应用是否在不使用的时候还保持连接打开？如果是这样，尤其是连接到很多服务器时，可能会过多地消耗其他进程所需要的连接。例如，假设你连接到 10 个 MySQL 服务器。从一个 Apache 进程中获取 10 个连接不是问题，但是任意时刻其中只有 1 个在真正工作。其他 9 个大部分时间都处于 Sleep 状态。如果其中一台服务器变慢了，或者有一个很长的网络请求，其他的服务器就可能因为连接数过多受到影响。解决方案是控制应用怎么使用连接。例如，可以将操作批量地依次发送到每个 MySQL 实例，并且在下一次执行 SQL 前关闭每个连接。如果执行的是比较消耗时间的操作，例如调用 Web 服务接口，甚至可以先关闭 MySQL 连接，执行耗时的工作，再重新打开 MySQL 连接继续在数据库上工作。

长连接和连接池的区别可能使人困惑。长连接可能跟连接池有同样的副作用，因为重用的连接在这两种情况下都是有状态的。

然而，连接池通常不会导致服务器连接过多，因为它们会在进程间排队和共享连接。另一方面，长连接是在每个进程基础上创建，不会在进程间共享。

连接池也比共享连接的方式对连接策略有更强的控制力。连接池可以配置为自动扩展，但是通常的实践经验是，当遇到连接池完全占满时，应该将连接请求进行排队而不是扩展连接池。这样做可以在应用服务器上进行排队等待，而不是将压力传递到 MySQL 数据库服务器上导致连接数太多而过载。

有很多方法可以使得查询和连接更快，但是一般的规则是，如果能够直接避免进行查询和连接，肯定比努力提升查询和连接的性能能获得更好的优化结果。

14.2 Web 服务器问题 608

Apache 是最流行的 Web 应用服务器软件。它在许多情况下都运行良好，但如果使用不当也会消耗大量的资源。最常见的问题是保持它的进程的存活（alive）时间过长，或者在各种不同的用途下混合使用，而不是分别对不同类型的工作进行优化。

Apache 通常是通过 prefork 配置来使用 `mod_php`、`mod_perl` 和 `mod_python` 模块的。prefork 模式会为每个请求预分配进程。因为 PHP、Perl 和 Python 脚本是可以定制化的，每个进程使用 50MB 或 100MB 内存的情况并不少见。当一个请求完成后，会释放大部分内存给操作系统，但并不是全部。Apache 会保持进程处于打开状态以备后来的请求重用。这意味着，如果下一个请求是请求静态文件，比如一个 CSS 文件或者一张图片，就

会出现用一个占用内存很多的进程来为一个很小的请求服务的情况。这就是使用 Apache 作为通用 Web 服务器很危险的原因。它的确是为通用目的而设计的，但如果能够有针对性地使用其长处，会获得更好的性能。

另一个主要的问题是，如果开启了 Keep-Alive 设置，进程可能很长时间处于繁忙状态。当然，即使没有开启 Keep-Alive，某些进程也可能存活很久，“填鸭式”地将内容传给客户端可能导致获取数据很慢[注1]。

人们常犯的另外一个错误，就是保持那些 Apache 默认开启的模块不动。

最好能够精简 Apache 的模块，移除掉那些不需要的。这很简单：只需要检查 Apache 的配置文件，注释掉不想要的模块，然后重启 Apache 就行。也可以在 *php.ini* 文件中删除不使用的 PHP 模块。

最差情况是，如果用一个通用目的的 Apache 配置直接用于 Web 服务，最后很可能产生很多重量级的 Apache 进程。这将浪费 Web 服务器的资源。它们还可能保持大量 MySQL 连接，浪费 MySQL 的资源。下面是一些可以降低服务器负载的方法[注2]。

不要使用 Apache 来做静态内容服务，或者至少和动态服务使用不同的 Apache 实例。流行的替代品有 Nginx（*http://www.nginx.com*）和 *lighttpd*（*http://www.lighttpd.net*）。

609
- 使用缓存代理服务器，比如 Squid 或者 Varnish，防止所有的请求都到达 Web 服务器。这个层面即使不能缓存所有页面，也可以缓存大部分页面，并且使用像 ESI（Edge Side Includes，参见 *http://www.esi.org*）这样的技术来将部分页面中的小块的动态内容嵌入到静态缓存部分。
- 对动态和静态资源都设置过期策略。可以使用 Squid 这样的缓存代理显式地使内容过期。维基百科就使用了这个技术来清理缓存中变更过的文章。

有时也许还需要修改应用程序，以便得到更长的过期时间。例如，如果你告诉浏览器永久缓存 CSS 和 JavaScript 文件，然后对站点的 HTML 做了一个修改，这个页面渲染将会出问题。这种情况可以为文件的每个版本设定唯一的文件名。例如，你可以定制网站的发布脚本，复制 CSS 文件到 */css/123_frontpage.css*，这里的 *123* 就是版本管理器中的版本号。对图片文件的文件名也可以这么做——永不重用文件名，这样页面就不会在升级时出问题，浏览器缓存多久的文件都没问题。

注 1： 填鸭式抓取发生在当一个客户端发起 HTTP 请求，但是没有迅速获取结果时。直到客户端获取整个结果，HTTP 连接——以及处理的 Apache 进程——都将保持活跃。

注 2： 有一本关于如何优化 Web 应用的很不错的书——*High Performance Web Sites*，作者是 Steve Souders (O'Reilly)。尽管书中大部分内容是从客户的角度来看如何让 Web 站点运行更快，但是参考他的建议也有利于你的服务器。Steve 后续的一本书 *Even Faster Web Sites* 也很不错，值得阅读。

- 不要让 Apache 填鸭式地服务客户端，这不仅仅会导致慢，也会导致 DDoS 攻击变得简单。硬件负载均衡器通常可以做缓冲，所以 Apache 可以快速地完成，让负载均衡器通过缓存响应客户端的请求，也可以在应用服务器前端使用 Nginx、Squid 或者事件驱动模式下的 Apache。
- 打开 *gzip* 压缩。对于现在的 CPU 而言这样做的代价很小，但是可以节省大部分流量。如果想节省 CPU 周期，可以使用缓存，或者诸如 Nginx 这样的轻量级服务器保存压缩过的页面版本。
- 不要为用于长距离连接的 Apache 配置启用 Keep-Alive 选项，因为这会使得重量级的 Apache 进程存活很长时间。可以用服务器端的代理来处理保持连接的工作，从而防止 Apache 被客户端拖垮。配置 Apache 到代理之间的连接使用 Keep-Alive 是可以的，因为代理只会使用很少的 Apache 连接去获取数据。图 14-1 展示了这个区别。

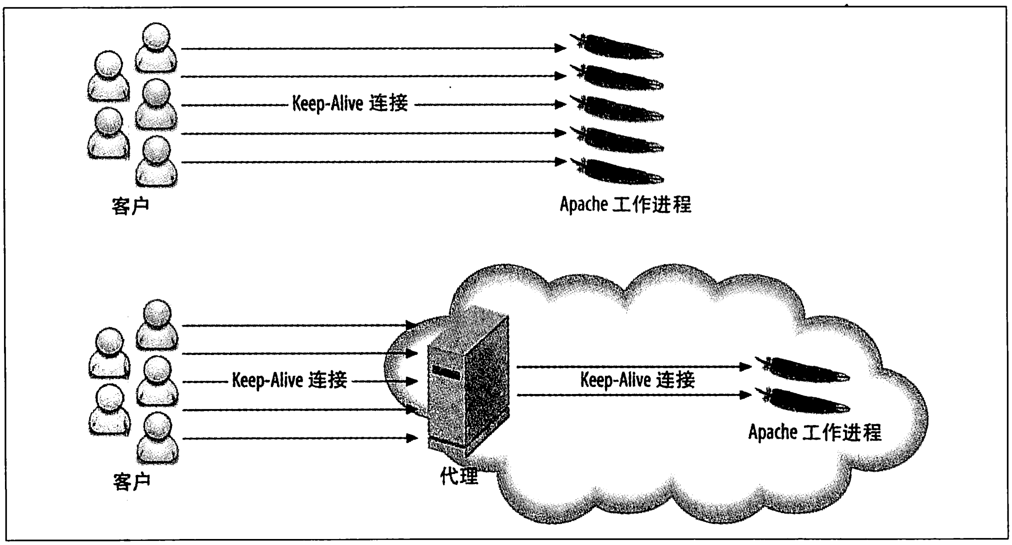


图14-1：代理可以使Apache不被长连接拖垮，产生更少的Apache工作进程。

这些策略可以使 Apache 进程存活时间变得很短，所以会有比实际需求更多的进程。无论如何，有些操作依然可能导致 Apache 进程存活时间太长，并且占用大量资源。举个例子，一个请求查询延时非常大的外部资源，例如远程的 Web 服务，就会出现 Apache 进程存活时间太长的问题。这种问题通常是无解的。

14.2.1 寻找最优并发度

每个 Web 服务器都有一个最佳并发度——就是说，让进程处理请求尽可能快，并且不超过系统负载的最优的并发连接数。这就是我们在第 11 章说的最大系统容量。进行一个简

单的测量和建模，或者只是反复试验，就可以找到这个“神奇的数”，为此花一些时间是值得的。

610 对于大流量的网站，Web 服务器同一时刻处理上千个连接是很常见的。然而，只有一小部分连接需要进程实时处理。其他的可能是读请求，处理文件上传，填鸭式服务内容，或者只是等待客户端的下一步请求。

随着并发的增加，服务器会逐渐到达它的最大吞吐量。在这之后，吞吐量通常开始降低。更重要的是，响应时间（延迟）也会因为排队而开始增加。

为什么会这样呢？试想，如果服务器只有一个 CPU，同时接收到了 100 个请求，会发生什么事情呢？假设 CPU 每秒能够处理一个请求。即便理想情况下操作系统没有调度的开销，也没有上下文切换的成本，那 100 个请求也需要 CPU 花费整整 100s 才能完成。

处理请求的最好方法是什么？可以将其一个个地排到队列中，也可以并行地执行并在不同请求之间切换，每次切换都给每个请求相同的服务时间。在这两种情况下，吞吐量都是每秒处理一个请求。然而，如果使用队列（并发 =1），平均延时是 50s，如果是并发执行（并发 =100）则是 100s。在实践中，并发执行会使平均延时更高，主要是因为上下文切换的代价。

对于 CPU 密集型工作负载，最佳并发度等于 CPU 数量（或者 CPU 核数）。然而，进程并不总是处于可运行状态的，因为会有一些阻塞式请求，例如 I/O、数据库查询，以及网络请求。因此，最佳并发度通常会比 CPU 数量高一些。

可以预测最优并发度，但是这需要精确的分析。尝试不同的并发值，看看在不增加响应
611 时间的情况下的最大吞吐量是多少，或者测量真正的工作负载并且进行分析，这通常更容易。Percona Toolkit 的 *pt-tcp-model* 工具可以帮助从 TCP 转储中测量和建模分析系统的可扩展性和性能特性。

14.3 缓存

缓存对高负载应用来说是至关重要的。一个典型的 Web 应用程序会提供大量的内容，直接生成这些内容的成本比采用缓存要高得多（包含检查和缓存超时的开销），所以采用缓存通常可以获得数量级的性能提升。诀窍是找到正确的粒度和缓存过期策略组合。另外也需要决定哪些内容适合缓存，缓存在哪里。

典型的高负载应用会有很多层缓存。缓存并不仅仅发生在服务器上，而是在每一个环节，甚至包括用户的 Web 浏览器（这就是内容过期头的用处）。通常，缓存越接近客户端，就越节省资源并且效率更高。从浏览器缓存提供一张图片比从 Web 服务器的内存获取快

得多，而从服务器的内存读取又比从服务器的磁盘上读取好得多。每种类型的缓存有其不一样的特点，例如容量和延时；在后面的章节我们会解释其中的一部分。

可以把缓存分成两大类：被动缓存和主动缓存。被动缓存除了存储和返回数据外不做任何事情。当从被动缓存请求一些内容时，要么可以得到结果，要么得到“结果不存在”。被动缓存的一个典型例子是 *memcached*。

相比之下，主动缓存会在访问未命中时做一些额外的工作。通常会将请求转发送给应用的其他部分来生成请求结果，然后存储该结果并返回给应用。Squid 缓存代理服务器就是一个主动缓存。

设计应用程序时，通常希望缓存是主动的（也可以叫做透明的），因为它们对应用隐藏了检查—生成—存储这个逻辑过程。也可以在被动缓存的前面构建一个主动缓存。

14.3.1 应用层以下的缓存

MySQL 服务器有自己的内部缓存，但也可以构建你自己的缓存和汇总表。可以对缓存表量身定制，使它们最有效地过滤、排序、与其他表关联、计数，或者用于其他用途。缓存表也比许多应用层缓存更持久，因为在服务器重启后它们还存在。

在第 4 章和第 5 章已经介绍了关于缓存策略的内容，所以在这一章，我们主要关注应用层以及更高层次的缓存。

缓存并不总是有用

612

必须确认缓存真的可以提升性能，因为有时缓存可能没有任何帮助。例如，在实践中发现从 Nginx 的内存中获取内容比从缓存代理中获取要快。如果代理的缓存在磁盘上则尤其如此。

原因很简单：缓存自身也有一些开销。比如检查缓存是否存在，如果命中则直接从缓存中返回数据。另外将缓存对象失效或者写入新的缓存对象都会有开销。缓存只在这些开销比没有缓存的情况下生成和提供数据的开销少时才有用。

如果知道所有这些操作的开销，就可以计算出缓存能提供多少帮助。没有缓存时的开销就是为每个请求生成数据的开销。有缓存时的开销是检查缓存的开销加上缓存不命中的概率乘以生成数据的开销，再加上缓存命中的概率乘以缓存提供数据的开销。

> 如果有缓存时的开销比没有时要低，则说明缓存可能有用，但依然不能保证。还要记住，就像从 Nginx 的内存中获取数据比从代理在磁盘中的缓存获取要好一样，有些缓存的开销比另外一些要低。

14.3.2 应用层缓存

应用层缓存通常在同一台机器的内存中存储数据，或者通过网络存在另一台机器的内存中。

因为应用可以缓存部分计算结果，所以应用层缓存可能比更低层次的缓存更有效。因此，应用层缓存可以节省两方面的工作：获取数据以及基于这些数据进行计算。一个很好的例子是 HTML 文本块。应用程序可以生成例如头条新闻的标题这样的 HTML 片段，并且做好缓存。后续的页面视图就可以简单地插入这个缓存过的文本。一般来说，在缓存数据前对数据做的处理越多，缓冲命中节省的工作就越多。

但应用层缓存也有缺点，那就是缓存命中率可能更低，并且可能使用较多的内存。假设需要 50 个不同版本的头条新闻标题，以使不同地区生活的用户看到不同的内容，那就需要足够的内存去存储全部 50 个版本，任何给定版本的标题命中次数都会更少，并且失效策略也会更加复杂。

应用缓存有许多种，下面是其中的一小部分。

本地缓存

这种缓存通常很小，只在进程处理请求期间存在于进程内存中。本地缓存可以有效地避免对某些资源的重复请求。这种类型的缓存技术并不复杂：通常只是应用代码中的一个变量或者哈希表。例如，假设需要显示一个用户名，而且已经知道其 ID，
613 就可以创建一个 `get_name_from_id()` 函数并且在其中增加缓存，像下面这样。

```
<?php
function get_name_from_id($user_id) {
   static $name; // static makes the variable persist
   if ( !$name ) {
      // Fetch name from database
   }
   return $name;
}
?>
```

如果使用的是 Perl，那么 Memoize 模块是函数调用结果标准的缓存方式。

```
use Memoize qw(memoize);
memoize 'get_name_from_id';
sub get_name_from_id {
   my ( $user_id ) = @_;
   my $name = # get name from database
   return $name;
}
```

这些技巧都很简单，但却可以为应用程序节省很多工作。

本地共享内存缓存

这种缓存一般是中等大小（几个 GB），快速，难以在多台机器间同步。它们对小型的半静态位数据比较合适。例如每个州的城市列表，分片数据存储的分区函数（映射表），或者使用存活时间（TTL）策略进行失效的数据等。共享内存最大的好处是访问非常快——通常比其他任何远程缓存访问都要快不少。

分布式内存缓存

最常见的分布式内存缓存的例子是 memcached。分布式缓存比本地共享内存缓存要大得多，增长也容易。缓存中创建的数据的每一个比特都只有一份副本，这样既不会浪费内存，也不会因为相同的数据存在不同的地方而引入一致性问题。分布式内存非常适合存储共享对象，例如用户资料、评论，以及 HTML 片段。

分布式缓存比本地共享缓存的延时要高得多，所以最高效的使用方法是批量进行多个获取操作（例如，在一次循环中获取多个对象）。分布式缓存还需要考虑怎么增加更多的节点，以及某个节点崩溃了怎么处理。对于这两个场景，应用程序必须决定在节点间怎么分布或重分布缓存对象。当缓存集群增加或减少一台服务器时，一致性缓存对避免性能问题而言是非常重要的。在下面这个网站有一个为 *memcached* 做的一致性缓存库：*http://www.audioscrobbler.net/development/ketama/*。

磁盘上的缓存 614

磁盘是很慢的，所以缓存在磁盘上的最好是持久化对象，很难全部装进内存的对象，或者静态内容（例如预处理的自定义图片）。

对于磁盘上的缓存和 Web 服务器，一个非常有用的技巧是使用 404 错误处理机制来捕捉缓存未命中的情况。假设 Web 应用要在头部展示一张基于用户名（“欢迎回来，John!”）的自定义图片，并且通过 */images/welcomeback/john.jpg* 这样的路径引用此图片。如果图片不存在，将会导致一个 404 错误，并且触发上述错误处理。这个错误处理可以生成图片，在磁盘上存储它，然后发出一个重定向或者将该图片传回浏览器。后续的请求只需要从文件中直接返回图片。

有很多类型的内容可以使用这种技巧。例如，不用再将最近的标题作为 HTML 部分进行缓存，可以在 JavaScript 文件中存储这些东西，然后在网页头中引用这个文件：*latest_headlines.js*。

缓存失效很简单：删除文件即可。可以通过执行一个删除 *N* 分钟前所创建的文件的

定时任务，来实现 TTL 失效。如果想要限制缓存大小，也可以通过按最近访问时间排序来删除文件，从而实现最近最少使用（LRU）失效算法。

如果失效策略是基于最近访问时间，则必须在文件系统挂载参数中打开访问时间记录。（忽略 `noatime` 选项即可。）如果这么做，应该使用内存文件系统来避免大量磁盘操作。

14.3.3 缓存控制策略

缓存也有像反范式化数据库设计一样的问题：重复数据，也就是说有多个地方需要更新数据，所以需要想办法避免读到脏数据。下面是一些最常见的缓存控制策略。

TTL（time to live，*存活时间*）

缓存对象存储时设置一个过期时间；可以通过清理进程在达到过期时间后删掉对象，或者先留着直到下次访问时再清理（清理后需要使用新的版本替换）。对于数据很少变更或者没有新数据的情况，这是最好的失效策略。

显式失效

如果不能接受脏数据，那么进程在更新原始数据时需要同时使缓存失效。这种策略有两个变种：写—失效和写—更新。写—失效策略很简单：只需要标记缓存数据已
615 经过期（是否清理缓存数据是可选的）。写—更新策略需要多做一些工作，因为在更新数据时就需要替换掉缓存项。无论如何，这都是非常有益的，特别是当生成缓存数据代价很昂贵时（写线程也许已经做了）。如果更新缓存数据，后续的请求将不再需要等待应用来生成。如果在后台做失效处理，例如基于 TTL 的失效，就可以在一个从用户请求完全分离出来的进程中生成失效数据的新版本。

读时失效

在更改旧数据时，为了避免要同时失效派生出来的脏数据，可以在缓存中保存一些信息，当从缓存中读数据时可以利用这些信息判断数据是否已经失效。和显式失效策略相比，这样做有很大的优势：成本固定且可以分散在不同时间内。假设要失效一个有一百万缓存对象依赖的对象，如果采用写时失效，需要一次在缓存中失效一百万个对象，即使有高效的方法来找到这些对象，也可能需要很长的时间才能完成。如果采用读时失效，写操作可以立即完成，但后续这一百万对象的读操作可能会有略微的延迟。这样就把失效一百万对象的开销分散了，并且可以帮助避免出现负载冲高和延迟增大的峰值。

一种最简单的读时失效的办法是采用对象版本控制。使用这种方法，在缓存中存储一个对象时，也可以存储对象所依赖的数据的当前版本号或者时间戳。例如，假设要缓存用户博客日志的统计信息，包括用户发表的博客数。当缓存 `blog_stats` 对象时，也可以同时存储用户的当前版本号，因为该统计信息是依赖于用户的。

不管什么时候更新依赖于用户的数据，都需要更新用户的版本号，假设用户的版本号初始为 0，并且由你来生成和缓存统计信息。当用户发表了一篇博客，就增加用户的版本号到 1（当然也要同时存储这篇博客，尽管在这个例子并没有用到博客数据）。然后当需要显示统计数据的时候，可以对缓存中 `blog_stats` 对象的版本与缓存的用户版本进行比较。因为用户的版本比对象的版本高，所以可以知道缓存的统计信息已经过期了，需要重新计算。

这是一个非常粗糙的内容失效方式，因为它假设依赖于用户的每一个比特的数据与所有其他数据都有交互。但这个假设并不总是成立的。举个例子，如果一个用户对一篇博客做了编辑，你也需要增加用户的版本号，这就会导致存储的统计信息失效，而实际上统计信息（发表的博客数）并没真的改变。这个取舍是很简单的。一个简单的缓存失效策略不只是更容易创建，也可能更加高效。

对象版本控制是一种简单的标记缓存方法，它可以处理更复杂的依赖关系。一个标记的缓存可以识别不同类型的依赖，并且分别跟踪每个依赖的版本。回到第 11 章中图书俱乐部的例子，你可以通过下面的版本号标记评论，使缓存的评论依赖于用户的版本和书的版本：user_ver=1234 和 book_ver=5678。任一版本号变了，都应该刷新缓存的评论。 616

14.3.4 缓存对象分层

分层缓存对象对检索、失效和内存利用都有帮助。相对于只缓存对象，也可以缓存对象的 ID、对象的 ID 组等通常需要一起检索的数据。

电子商务网站的搜索结果是这种技术很好的例子。一次搜索可能返回一个匹配产品的列表，包括名称、描述、缩略图，以及价格。缓存整个列表的效率很低：其他的搜索也可能会包含一些相同的产品，这就会导致数据重复，并且浪费内存。这种策略也使得当一个产品的价格变动时，找出并失效搜索结果变得很困难，因为你必须查看每个列表，找到哪些列表包含了更新过的产品。

可以缓存关于搜索的最小信息，而不必缓存整个列表，例如返回结果的数量以及列表中的产品 ID。然后可以再单独缓存每个产品。这样做可解决两个问题：不会重复存放任何结果数据，也更容易在失效产品的粒度上去失效缓存。

缺点则是，相对于一次性获得整个搜索结果，必须在缓存中检索多个对象。然而不管怎么说，为搜索结果缓存产品 ID 的列表都是更有效的做法。先在一个缓存命中返回 ID 的列表，再使用这些 ID 去请求缓存获得产品信息。如果缓存允许在一次调用里返回多个结果，第二次请求就可以返回多个产品（*memcached* 通过 `mget()` 调用来支持）。

如果使用不当，这种方法可能会导致奇怪的结果。假设使用 TTL 策略来失效搜索结果，并且当产品变更时显式地去失效单个产品。现在想象一下，一个产品的描述发生了变化，不再包含搜索中匹配的关键字，但是搜索结果的缓存还没有过期失效。此时用户就会看到错误的搜索结果，因为缓存的搜索结果将会引用这个变化了的产品，即使它不再包含匹配搜索的关键字。

对于大多数应用程序来说，这不是问题。如果应用程序不能容忍这种情况，可以使用基于版本的缓存，并在执行搜索时在结果中存储产品的版本号。当发现搜索结果在缓存中时，可以将当前搜索结果的版本号和搜索结果中每个产品的版本号做比较。如果发现任何一个产品的版本数据不一致，可以重新搜索并且重新缓存结果。

这对理解远程缓存访问的花销是多么昂贵非常重要。虽然缓存很快，也可以避免很多工作，但在 LAN 环境下网络往返缓存服务器通常也需要 0.3ms 左右。我们见过很多案例，
617 复杂的网页需要一千次左右的缓存访问来组合页面结果，这将会耗费 3s 左右的网络延时，意味着你的页面可能慢得不可接受，即使它甚至不需要访问数据库！因此，在这种情况下对缓存使用批量获取调用是非常重要的。对缓存进行分层，采用小一些的本地缓存，也可能获得很大的收益。

14.3.5 预生成内容

除了在应用程序级别缓存位数据，也可以在后台预先请求一些页面，并且将结果存为静态页面。如果页面是动态的，也可以预先生成页面的部分内容，然后使用像服务端包含（SSI）这样的技术创建最终页面。这有助于减小预生成内容的大小和开销，否则可能在将不同部分拼装到最终页面的时候，由于微小的变化产生大量的重复内容。几乎可以对任何类型的缓存使用预生成策略，包括 *memcached*。

预生成内容有几个重要的好处。

- 应用代码没有复杂的命中和未命中处理路径。
- 当未命中的处理路径慢得不可接受时，这种方案可以很好地工作，因为它保证了未命中的情况永远不会发生。实际上，在任何时候设计任何类型的缓存系统，总是应该考虑未命中的路径有多慢。如果平均性能提升很大，但是因为要预生成缓存内容，偶尔有一些请求变得非常缓慢，这时可能比不用缓存还糟糕。性能的持续稳定通常跟高性能一样重要。
- 预生成内容可以避免在缓存未命中时导致的雪崩效应。

缓存预生成好的内容可能占用大量空间，并且并不总能预生成所有东西。无论是哪种形式的缓存，需要预生成的内容中最重要的部分是那些最经常被请求，或者生成的成本最

高的，所以可以通过本章前面提到的 404 错误处理机制来按需生成。

预生成的内容有时候也可以从内存文件系统中获益，因为可以避免磁盘 I/O。

14.3.6 作为基础组件的缓存

缓存有可能成为基础设施的重要组成部分。也很容易陷入一个陷阱，认为缓存虽然很好用，但并不是重要到非有不可的东西。你也许会辩驳，如果缓存服务器宕机或者缓存被清空，请求也可以直接落在数据库上，系统依然可以正常运行。如果是刚刚将缓存加入应用系统，这也许是对的，但是缓存的加入可以使得在应用压力显著增长时不需要对系统的某些部分同比增加资源投入——通常是数据库部分。因此，系统可能慢慢地变得对缓存非常依赖，却没有被发觉。

例如，如果高速缓存命中率是 90%，当由于某种原因失去缓存，数据库上的负载将增加到原来的 10 倍。这很可能导致压力超过数据库服务器的性能极限。 618

为了避免像这样的意外，应该设计一些高可用性缓存（包括数据和服务）的解决方案，或者至少是评估好禁用缓存或丢失缓存时的性能影响。比如说可以设计应用在遇到这样的情况时能够进行降级处理。

14.3.7 使用 HandlerSocket 和 memcached

相对于数据存储在 MySQL 中而缓存在 MySQL 外部的缓存方案，另外有一种替代方法是为 MySQL 创建一个更快的访问路径，直接绕过使用缓存。对于小而简单的查询语句，很大一部分开销来自解析 SQL，检查权限，生成执行计划，等等。如果这种开销可以避免，MySQL 在处理简单查询时将非常快。

目前有两个解决方案可以用所谓的 NoSQL 方式访问 MySQL。第一种是一个后台进程插件，称为 HandlerSocket，由 DeNA 开发，这是日本最大的社交网站。HandlerSocket 允许通过一个简单的协议访问 InnoDB Handler 对象。实际上，也就是绕过了上层的服务器层，通过网络直接连接到了 InnoDB 引擎层。有报告称 HandlerSocket 每秒可以执行超过 750 000 条查询。Percona Server 分支中自带了 HandlerSocket 插件引擎层。

第二个方案是通过 *memcached* 协议访问 InnoDB。MySQL 5.6 的实验室版本有一个插件提供了这个接口。

两种方法都有一些限制——特别是 *memcached* 的方法，这种方法对很多访问数据的方法都不支持。为什么会希望采用 SQL 以外的什么办法访问数据呢？除了速度之外，最大的原因可能是简单。这样做最大的好处是可以摆脱缓存，以及所有的失效逻辑，还有为它们服务的额外的基础设施。

14.4 拓展 MySQL

如果 MySQL 不能做你需要的事，一种可能是拓展其功能。在这里我们不会展示如何做到这一点，但会提供一些可能的方向。如果你对进一步探索有兴趣，那么有很多很好的在线资源，以及许多关于这些内容的书籍可以参考。

当我们说“MySQL 不能做你需要的事”，我们指的是两件事情：MySQL 根本做不到这一点，或者 MySQL 可以做到，但是只能通过缓慢或笨拙的方法，总之做得不够好。无论哪个都是需要对 MySQL 拓展的原因。好消息是，MySQL 已经越来越模块化和通用。

619 存储引擎是拓展 MySQL 的一个很好的方式。Brian Aker 已经写了一个存储引擎的框架，还有一系列介绍有关如何开始编写自己的存储引擎的文章。这是目前几个主要的第三方存储引擎的基础。许多公司都编写了它们自己的内部存储引擎。例如，一些社交网络公司使用了特殊的为社交图形操作设计的存储引擎，我们还知道有个公司定制了一个用于模糊搜索的引擎。写一个简单的自定义存储引擎并不难。

还可以使用存储引擎作为另一个软件的接口。Sphinx 引擎就是一个很好的例子，该引擎是 Sphinx 全文检索软件的接口（见附录 F）。

14.5 MySQL 的替代品

MySQL 并不是适合每一个场景的解决方案。有些工作通常在 MySQL 以外来做会更好，即使 MySQL 理论上也可以做到。

最明显的一个例子是在传统的文件系统中存储文件，而不是在表中。图像文件是经典案例：虽然可以把它们放到一个 BLOB 列，但这通常不是个好办法[注3]。一般的做法是，在文件系统中存储图片或其他大型二进制文件，而在 MySQL 中只存储文件名；然后应用程序在 MySQL 之外存取文件。对于 Web 应用程序，可以把文件名放在 `<img>` 元素的 `src` 属性中，这样就可以实现对文件的存取。

全文检索是另一个最好放在 MySQL 之外处理的例子——MySQL 在全文搜索方面明显不如 Lucene 和 Sphinx。

NDB API 也可能对某些任务有用。例如，尽管 MySQL 的 NDB 集群存储引擎（目前还）不适合存储一个高性能 Web 应用程序的全部数据，但用 NDB API 直接存储网站会话数据或用户注册信息还是可能的。在如下网站可以了解到更多关于 NDB API 的内容：*http://dev.mysql.com/doc/ndbapi/en/index.html*。还有供 Apache 使用的 NDB 模块，*mod_*

注 3： 使用 MySQL 的复制来快速分布镜像到其他机器更有优势，我们知道一些程序使用这种技术。

ndb，可以在 *http://code.google.com/p/mod-ndb/* 下载。

最后，对于某些操作——如图形关系和树遍历——关系型数据库并不总是正确的典范。MySQL 并不擅长分布式数据处理，因为它缺乏并行执行查询的能力。出于这些目的情况还是建议使用其他工具（可能与 MySQL 结合）。现在想到的例子包括：

- 对于简单的键一值存储，在复制严重落后的非常高速的访问场景中，我们建议用 Redis 替换 MySQL。即使 MySQL 主库可以承担这样的压力，备库的延迟也是非常让人头疼的。Redis 也常用来做队列，因为它对队列操作支持得很好。 620
- Hadoop 是房间中的大象，一语双关。混合 MySQL/Hadoop 的部署在处理大型或半结构化数据时非常常见。

14.6 总结

优化并不只是数据库的事。正如我们在第 3 章建议的，最高形式的优化既包含业务上的，也包含用户层的。全方位的优化才是好的优化。

一般来说，首先要做的事是测量。认真剖析每一层的问题。哪一层导致了大部分的响应时间？对这一层就要重点关注。如果用户的经验是大部分的时间消耗在浏览器的 DOM 渲染上面，MySQL 只贡献总响应时间的一小部分，那么进一步优化查询语句绝对不可能明显地改善用户体验。在测量完成后，通常很容易理解应该在哪里投入精力。我们建议阅读 Steve Souders 的两本书（*High Performance Web Sites* 和 *Even Faster Web Sites*），并且建议使用 New Relic 工具。

在 Web 服务器的配置和缓存中经常可以发现大问题，而这些问题往往很容易解决。还有一个固有的观念，“总是数据库的问题”，但这其实是不正确的。应用程序中的其他层也同样重要，它们很可能被错误配置，尽管有时不太明显。特别是缓存，能承受比只使用 MySQL 要低得多的成本传递大量内容。虽然 Apache 依然是世界上最流行的 Web 服务器软件，但它并不总是最合适的工具，因此考虑像 Nginx 这样的替代方案也是非常有意义的。

第 15 章 621

备份与恢复

如果没有提前做好备份规划，也许以后会发现已经错失了一些最佳的选择。例如，在服务器已经配置好以后，才想起应该使用 LVM，以便可以获取文件系统的快照——但这时已经太迟了。在为备份配置系统参数时，可能没有注意到某些系统配置对性能有着重要影响。如果没有计划做定期的恢复演练，当真的需要恢复时，就会发现并没有那么顺利。

相对于本书的第一版和第二版来说，我们在此假设大部分用户主要使用 InnoDB 而不是 MyISAM。在本章中，我们不会涵盖一个精心设计的备份和恢复解决方案的所有部分——而仅涉及与 MySQL 相关的部分。我们不打算包括的话题如下 ：

- 安全（访问备份，恢复数据的权限，文件是否需要加密）。
- 备份存储在哪里，包括它们应该离源数据多远（在一块不同的盘上，一台不同的服务器上，或离线存储），以及如何将数据从源头移动到目的地。
- 保留策略、审计、法律要求，以及相关的条款。
- 存储解决方案和介质，压缩，以及增量备份。
- 存储的格式。
- 对备份的监控和报告。
- 存储层内置备份功能，或者其他专用设备，例如预制式文件服务器。

像这样的话题已经在许多书中涉及，例如 W. Curtis Preston 的 *Backup & Recouery*（O'Reilly）。

在开始本章之前，让我们先澄清几个核心术语。首先，经常可以听到所谓的热备份、暖备份和冷备份。人们经常使用这些词来表示一个备份的影响 ：例如，“热”备份不需要任何的服务停机时间。问题是对这些术语的理解因人而异。有些工具虽然在名字中使用

622 了“热备份”，但实际上并不是我们所认为的那样。我们尽量避开这些术语，而直接说明某个特别的技术或工具对服务器的影响。

另外两个让人困惑的词是还原和恢复。在本章中它们有其特定的含义。还原意味着从备份文件中获取数据，可以加载这些文件到 MySQL 里，也可以将这些文件放置到 MySQL 期望的路径中。恢复一般意味着当某些异常发生后对一个系统或其部分的拯救。包括从备份中还原数据，以及使服务器完全恢复功能的所有必要步骤，例如重启 MySQL、改变配置和预热服务器的缓存等。

在很多人的概念中，恢复仅意味着修复崩溃后损坏的表。这与恢复一个完整的服务器是不同的。存储引擎的崩溃恢复要求数据和日志文件一致。要确保数据文件中只包含已经提交的事务所做的修改，恢复操作会将日志中还没有应用到数据文件的事务重新执行。这也许是恢复过程的一部分，甚至是备份的一部分。然而，这和一个意外的 DROP TABLE 事故后需要做的事是不一样的。

15.1 为什么要备份

下面是备份非常重要的几个理由：

灾难恢复

灾难恢复是下列场景下需要做的事情：硬件故障、一个不经意的 Bug 导致数据损坏，或者服务器及其数据由于某些原因不可获取或无法使用等。你需要准备好应付很多问题：某人偶然连错服务器执行了一个 ALTER TABLE[注1] 的操作，机房大楼被烧毁，恶意的黑客攻击或MySQL的Bug等。尽管遭受任何一个特殊的灾难的几率都非常低，但所有的风险叠加在一起就很有可能会碰到。

人们改变想法

不必惊讶，很多人经常会在删除某些数据后又想要恢复这些数据。

审计

有时候需要知道数据或 Schema 在过去的某个时间点是什么样的。例如，你也许被卷入一场法律官司，或发现了应用的一个 Bug，想知道这段代码之前干了什么（有时候，仅仅依靠代码的版本控制还不够）。

623 测试

一个最简单的基于实际数据来测试的方法是，定期用最新的生产环境数据更新测试服务器。如果使用备份的方案就非常简单：只要把备份文件还原到测试服务器上即可。

注 1： Baron 仍然记得他毕业后的第一个工作，当时他把电子商务网站的生产服务器上的发货表删除了两列。

检查你的假设。例如，你认为共享虚拟主机供应商会提供 MySQL 服务器的备份？许多主机供应商根本不备份 MySQL 服务器，另外一些也仅仅在服务器运行时复制文件，这可能会创建一个损坏的没有用处的备份。

15.2 定义恢复需求

如果一切正常，那么永远也不需要考虑恢复。但是，一旦需要恢复，只有世界上最好的备份系统是没用的，还需要一个强大的恢复系统。

不幸的是，让备份系统平滑工作比构造良好的恢复过程和工具更容易。原因如下：

- 备份在先。只有已经做了备份才可能恢复，因此在构建系统时，注意力自然会集中在备份上。
- 备份由脚本和任务自动完成。经常不经意地，我们会花些时间调优备份过程。花 5 分钟来对备份过程做小的调整看起来并不重要，但是你是否天天同样地重视恢复呢？
- 备份是日常任务，但恢复常常发生在危急情形下。
- 因为安全的需要，如果正在做异地备份，可能需要对备份数据进行加密，或采取其他措施来进行保护。安全性往往只关注数据被盗用的后果，但是有没有人想过，如果没有人能对用来恢复数据的加密卷解锁，或需要从一个整块的加密文件中抽取单个文件时，损害又是多大？
- 只有一个人来规划、设计和实施备份。当灾难袭来时，那个人可能不在。因此需要培养几个人并有计划地互为备份，这样就不会要求一个不合格的人来恢复数据。

这里有一个我们看到的真实例子：一个客户报告说当 *mysqldump* 加上 *-d* 选项后，备份变得像闪电一般快，他想知道为什么没有一个人提出该选项可以如此快地加速备份过程。如果这个客户已经尝试还原这些备份，就不难发现其原因：使用 *-d* 选项将不会备份数据！这个客户关注备份，却没有关注恢复，因此完全没有意识到这个问题。

规划备份和恢复策略时，有两个重要的需求可以帮助思考：恢复点目标（PRO）和恢复时间目标（RTO）。它们定义了可以容忍丢失多少数据，以及需要等待多久将数据恢复。在定义 RPO 和 RTO 时，先尝试回答下面几类问题： 624

- 在不导致严重后果的情况下，可以容忍丢失多少数据？需要故障恢复，还是可以接受自从上次日常备份后所有的工作全部丢失？是否有法律法规的要求？
- 恢复需要在多长时间内完成？哪种类型的宕机是可接受的？哪种影响（例如，部分服务不可用）是应用和用户可以接受的？当那些场景发生时，又该如何持续服务？
- 需要恢复什么？常见的需求是恢复整个服务器，单个数据库，单个表，或仅仅是特定的事务或语句。

建议将上面这些问题的答案明确地用文档记录下来，同时还应该明确备份策略，以及备份过程。

> **备份误区 1："复制就是备份"**
>
> 这是我们经常碰到的一个误区。复制不是备份，当然使用 RAID 阵列也不是备份。为什么这么说？可以考虑一下，如果意外地在生产库上执行了 DROP DATABASE，它们是否可以帮你恢复所有的数据？RAID 和复制连这个简单的测试都没法通过。它们不是备份，也不是备份的替代品。只有备份才能满足备份的要求。

15.3 设计 MySQL 备份方案

备份 MySQL 比看起来难。最基本的，备份仅是数据的一个副本，但是受限于应用程序的要求、MySQL 的存储引擎架构，以及系统配置等因素，会让复制一份数据都变得很困难。

在深入所有选项细节之前，先来看一下我们的建议：

- 在生产实践中，对于大数据库来说，物理备份是必需的：逻辑备份太慢并受到资源限制，从逻辑备份中恢复需要很长时间。基于快照的备份，例如 Percona XtraBackup 和 MySQL Enterprise Backup 是最好的选择。对于较小的数据库，逻辑备份可以很好地胜任。
- 保留多个备份集。
- 定期从逻辑备份（或者物理备份）中抽取数据进行恢复测试。
- 保存二进制日志以用于基于故障时间点的恢复。expire_logs_days 参数应该设置得足够长，至少可以从最近两次物理备份中做基于时间点的恢复，这样就可以在保持主库运行且不应用任何二进制日志的情况下创建一个备库。备份二进制日志与过期设置无关，二进制日志备份需要保存足够长的时间，以便能从最近的逻辑备份进行恢复。 625
- 完全不借助备份工具本身来监控备份和备份的过程。需要另外验证备份是否正常。
- 通过演练整个恢复过程来测试备份和恢复。测算恢复所需要的资源（CPU、磁盘空间、实际时间，以及网络带宽等）。
- 对安全性要仔细考虑。如果有人能接触生产服务器，他是否也能访问备份服务器？反过来呢？

弄清楚 RPO 和 RTO 可以指导备份策略。是需要基于故障时间点的恢复能力，还是从昨晚的备份中恢复但会丢失此后的所有数据就足够了？如果需要基于故障时间点的恢复，可能要建立日常备份并保证所需要的二进制日志是有效的，这样才能从备份中还原，并通过重放二进制日志来恢复到想要的时间点。

一般说来，能承受的数据丢失越多，备份越简单。如果有非常苛刻的需求，要确保能恢复所有数据，备份就很困难。基于故障时间点的恢复也有几类。一个“宽松”的故障时间点恢复需求意味着需要重建数据，直到“足够接近”问题发生的时刻。一个“硬性”的需求意味着不能容忍丢失任何一个已提交的事务，即使某些可怕的事情发生（例如服务器着火了）。这需要特别的技术，例如将二进制日志保存在一个独立的 SAN 卷或使用 DRBD 磁盘复制。

15.3.1 在线备份还是离线备份

如果可能，关闭 MySQL 做备份是最简单最安全的，也是所有获取一致性副本的方法中最好的，而且损坏或不一致的风险最小。如果关闭了 MySQL，就根本不用关心 InnoDB 缓冲池中的脏页或其他缓存。也不需要担心数据在尝试备份的过程被修改，并且因为服务器不对应用提供访问，所以可以更快地完成备份。

尽管如此，让服务器停机的代价可能比看起来要更昂贵。即使能最小化停机时间，在高负载和高数据量下关闭和重启 MySQL 也可能要花很长一段时间，这在第 8 章中讨论过。我们演示过一些使这个影响最小化的技术，但并不能将其减少为零。因此，必须要设计不需要生产服务器停机的备份。即便如此，由于一致性的需要，对服务器进行在线备份仍然会有明显的服务中断。

在众多的备份方法中，一个最大问题就是它们会使用 FLUSH TABLES WITH READ LOCK 操作。这会导致 MySQL 关闭并锁住所有的表，将 MyISAM 的数据文件刷新到磁盘上（但 InnoDB 不是这样的！），并且刷新查询缓存。该操作需要非常长的时间来完成。具体需要多长时间是不可预估的；如果全局读锁要等待一个长时间运行的语句完成，或有许多表，那么时间会更长。除非锁被释放，否则就不能在服务器上更改任何数据，一切都会被阻塞和积压[注2]。FLUSH TABLES WITH READ LOCK 不像关闭服务器的代价那么高，因为大部分缓存仍然在内存中，并且服务器一直是“预热”的，但是它也有非常大的破坏性。如果有人说这样做很快，可能是准备向你推销某种从来没有在真正的线上服务器上运行过的东西。 626

注 2： 是的，即使 SELECT 查询也会被阻塞，因为如果有一个查询需要修改某些数据，只要它开始等待表上的写锁，所有尝试获取读锁的查询也必须等待。

避免使用 FLUSH TABLES WITH READ LOCK 的最好的方法是只使用 InnoDB 表。在权限和其他系统信息表中使用 MyISAM 表是不可避免的，但是如果数据改变量很少（正常情况下），你可以只刷新和锁住这些表，这不会有什么问题。

在规划备份时，有一些与性能相关的因素需要考虑。

锁时间
: 需要持有锁多长时间，例如在备份期间持有的全局 FLUSH TABLES WITH READ LOCK？

备份时间
: 复制备份到目的地需要多久？

备份负载
: 在复制备份到目的地时对服务器性能的影响有多少？

恢复时间
: 把备份镜像从存储位置复制到 MySQL 服务器，重放二进制日志等，需要多久？

最大的权衡是备份时间与备份负载。可以牺牲其一以增强另外一个。例如，可以提高备份的优先级，代价是降低服务器性能。

同样，也可以利用负载的特性来设计备份。例如，如果服务器在晚上的 8 小时内仅仅有 50% 的负载，那么可以尝试规划备份，使得服务器的负载低于 50% 且仍能在 8 小时内完成。可以采用许多方法来完成这个目标，例如，可以用 *ionice* 和 *nice* 来提高复制或压缩操作的优先级，使用不同的压缩等级，或在备份服务器上压缩而不是在 MySQL 服务器上。
627 甚至可以利用 *lzo* 或 *pigz* 以获取更快的压缩。也可以使用 O_DIRECT 或 fadvise() 在复制操作时绕开操作系统的缓存，以避免污染服务器的缓存。像 Percona XtraBackup 和 MySQL Enterprise Backup 这样的工具都有限流选项，可在使用 *pv* 时加 *--rate-limit* 选项来限制备份脚本的吞吐量。

15.3.2 逻辑备份还是物理备份

有两种主要的方法来备份 MySQL 数据：逻辑备份（也叫"导出"）和直接复制原始文件的物理备份。逻辑备份将数据包含在一种 MySQL 能够解析的格式中，要么是 SQL，要么是以某个符号分隔的文本[注3]。原始文件是指存在于硬盘上的文件。

任何一种备份都有其优点和缺点。

注 3： 由 *mysqldump* 生成的逻辑备份并不一定是文本文件。SQL 导出会包含许多不同的字符集，同样也会包含二进制数据，这些数据并不是有效的字符。对于许多编辑器来说，文件行也可能会太长。但是，大多数这样的文件还是可以被编辑器打开和读取，特别是 *mysqldump* 使用了 *--hex-blob* 选项时。

逻辑备份

逻辑备份有如下优点：

- 逻辑备份是可以用编辑器或像 *grep* 和 *sed* 之类的命令查看和操作的普通文件。当需要恢复数据或只想查看数据但不恢复时，这都非常有帮助。
- 恢复非常简单。可以通过管道把它们输入到 *mysql*，或者使用 *mysqlimport*。
- 可以通过网络来备份和恢复——就是说，可以在与 MySQL 主机不同的另外一台机器上操作。
- 可以在类似 Amazon RDS 这样不能访问底层文件系统的系统中使用。
- 非常灵活，因为 *mysqldump*——大部分人喜欢的工具——可以接受许多选项，例如可以用 `WHERE` 子句来限制需要备份哪些行。
- 与存储引擎无关。因为是从 MySQL 服务器中提取数据而生成，所以消除了底层数据存储和不同。因此，可以从 InnoDB 表中备份，然后只需极小的工作量就可以还原到 MyISAM 表中。而对于原始数据却不能这么做。
- 有助于避免数据损坏。如果磁盘驱动器有故障而要复制原始文件时，你将会得到一个错误并且 / 或生成一个部分或损坏的备份。如果 MySQL 在内存中的数据还没有损坏，当不能得到一个正常的原始文件复制时，有时可以得到一个可以信赖的逻辑备份。

尽管如此，逻辑备份也有它的缺点： 628

- 必须由数据库服务器完成生成逻辑备份的工作，因此要使用更多的 CPU 周期。
- 逻辑备份在某些场景下比数据库文件本身更大[注4]。ASCII 形式的数据不总是和存储引擎存储数据一样高效。例如，一个整型需要 4 字节来存储，但是用 ASCII 写入时，可能需要 12 个字符。当然也可以压缩文件以得到一个更小的备份文件，但这样会使用更多的 CPU 资源。（如果索引比较多，逻辑备份一般要比物理备份小。）
- 无法保证导出后再还原出来的一定是同样的数据。浮点表示的问题、软件 Bug 等都会导致问题，尽管非常少见。
- 从逻辑备份中还原需要 MySQL 加载和解释语句，转化为存储格式，并重建索引，所有这一切会很慢。

最大的缺点是从 MySQL 中导出数据和通过 SQL 语句将其加载回去的开销。如果使用逻辑备份，测试恢复需要的时间将非常重要。

Percona Server 中包含的 *mysqldump*，在使用 InnoDB 表时能起到帮助作用，因为它会对输出格式化，以便在重新加载时利用 InnoDB 的快速建索引的优点。我们的测试显示这样做可以减少 2/3 甚至更多的还原时间。索引越多，好处越明显。

注 4： 以我们的经验，逻辑备份往往比物理备份要小许多，但也并不总是如此。

物理备份

物理备份有如下好处：

- 基于文件的物理备份，只需要将需要的文件复制到其他地方即可完成备份。不需要其他额外的工作来生成原始文件。
- 物理备份的恢复可能就更简单了，这取决于存储引擎。对于 MyISAM，只需要简单地复制文件到目的地即可。对于 InnoDB 则需要停止数据库服务，可能还要采取其他一些步骤。
- InnoDB 和 MyISAM 的物理备份非常容易跨平台、操作系统和 MySQL 版本。（逻辑导出亦如此。这里特别指出这一点是为了消除大家的担心。）
- 从物理备份中恢复会更快，因为 MySQL 服务器不需要执行任何 SQL 或构建索引。
629 如果有很大的InnoDB表，无法完全缓存到内存中，则物理备份的恢复要快非常多——至少要快一个数量级。事实上，逻辑备份最可怕的地方就是不确定的还原时间。

物理备份也有其缺点，比如：

- InnoDB 的原始文件通常比相应的逻辑备份要大得多。InnoDB 的表空间往往包含很多未使用的空间。还有很多空间被用来做存储数据以外的用途（插入缓冲，回滚段等）。
- 物理备份不总是可以跨平台、操作系统及 MySQL 版本。文件名大小写敏感和浮点格式是可能会遇到麻烦。很可能因浮点格式不同而不能移动文件到另一个系统（虽然主流处理器都使用 IEEE 浮点格式。）

物理备份通常更加简单高效[注5]。尽管如此，对于需要长期保留的备份，或者是满足法律合规要求的备份，尽量不要完全依赖物理备份。至少每隔一段时间还是需要做一次逻辑备份。

除非经过测试，不要假定备份（特别是物理备份）是正常的。对 InnoDB 来说，这意味着需要启动一个 MySQL 实例，执行 InnoDB 恢复操作，然后运行 CHECK TABLES。也可以跳过这一操作，仅对文件运行 *innochecksum*，但我们不建议这样做。对于 MyISAM，可以运行 CHECK TABLES，或者使用 *mysqlcheck*。使用 *mysqlcheck* 可以对所有的表执行 CHECK TABLES 操作。

建议混合使用物理和逻辑两种方式来做备份：先使用物理复制，以此数据启动 MySQL 服务器实例并运行 *mysqlcheck*。然后，周期性地使用 *mysqldump* 执行逻辑备份。这样做可以获得两种方法的优点，不会使生产服务器在导出时有过度负担。如果能够方便地利用文件系统的快照，也可以生成一个快照，将该快照复制到另外一个服务器上并释放，然后测试原始文件，再执行逻辑备份。

注 5： 值得一提的是物理备份会更易出错；很难像 *mysqldump* 一样简单。

15.3.3 备份什么

恢复的需求决定需要备份什么。最简单的策略是只备份数据和表定义，但这是一个最低的要求。在生产环境中恢复数据库一般需要更多的工作。下面是 MySQL 备份需要考虑的几点。

非显著数据

不要忘记那些容易被忽略的数据：例如，二进制日志和 InnoDB 事务日志。

代码 630

现代的 MySQL 服务器可以存储许多代码，例如触发器和存储过程。如果备份了 `mysql` 数据库，那么大部分这类代码也备份了，但如果需要还原单个业务数据库会比较麻烦，因为这个数据库中的部分"数据"，例如存储过程，实际是存放在 `mysql` 数据库中的。

复制配置

如果恢复一个涉及复制关系的服务器，应该备份所有与复制相关的文件，例如二进制日志、中继日志、日志索引文件和 *.info* 文件。至少应该包含 `SHOW MASTER STATUS` 和 / 或 `SHOW SLAVE STATUS` 的输出。执行 `FLUSH LOGS` 也非常有好处，可以让 MySQL 从一个新的二进制日志开始。从日志文件的开头做基于故障时间点的恢复要比从中间更容易。

服务器配置

假设要从一个实际的灾难中恢复，比如说，地震过后在一个新数据中心中构建服务器，如果备份中包含服务器配置，你一定会喜出望外。

选定的操作系统文件

对于服务器配置来说，备份中对生产服务器至关重要的任何外部配置，都十分重要。在 UNIX 服务器上，这可能包括 *cron* 任务、用户和组的配置、管理脚本，以及 *sudo* 规则。

这些建议在许多场景下会被当作"备份一切"。然而，如果有大量的数据，这样做的开销将非常高，如何做备份，需要更加明智的考虑。特别是，可能需要在不同备份中备份不同的数据。例如，可以单独地备份数据、二进制日志和操作系统及系统配置。

增量备份和差异备份

当数据量很庞大时，一个常见的策略是做定期的增量或差异备份。它们之间的区别有点容易让人混淆，所以先来澄清这两个术语：差异备份是对自上次全备份后所有改变的部分而做的备份，而增量备份则是自从任意类型的上次备份后所有修改做的备份。

例如，假如在每周日做一个全备份。在周一，对自周日以来所有的改变做一个差异备份。

在周二，就有两个选择：备份自周日以来所有的改变（差异），或只备份自从周一备份后所有的改变（增量）。

增量和差异备份都是部分备份：它们一般不包含完整的数据集，因为某些数据几乎肯定没有改变。部分备份对减少服务器开销、备份时间及备份空间而言都很适合。尽管
631 某些部分备份并不会真正减少服务器的开销。例如，Percona XtraBackup 和 MySQL Enterprise Backup，仍然会扫描服务器上的所有数据块，因而并不会节约太多的开销，但它们确实会减少一定量的备份时间和大量用于压缩的 CPU 时间，当然也会减少磁盘空间使用[注6]。

不要因为会用高级备份技术而自负，解决方案越复杂，可能面临的风险也越大。要注意分析隐藏的危险，如果多次迭代备份紧密地耦合在一起，则只要其中的一次迭代备份有损坏，就可能会导致所有的备份都无效。

下面有一些建议：

- 使用 Percona XtraBackup 和 MySQL Enterprise Backup 中的增量备份特性。
- 备份二进制日志。可以在每次备份后使用 `FLUSH LOGS` 来开始一个新的二进制日志，这样就只需要备份新的二进制日志。
- 不要备份没有改变的表。有些存储引擎，例如 MyISAM，会记录每个表最后修改时间。可以通过查看磁盘上的文件或运行 `SHOW TABLE STATUS` 来看这个时间。如果使用 InnoDB，可以利用触发器记录修改时间到一个小的“最后修改时间”表中，帮助跟踪最新的修改操作。需要确保只对变更不频繁的表进行跟踪，这样才能降低开销。通过定制的备份脚本可以轻松获取到哪些表有变更。
  例如，如果有包含不同语种各个月的名称列表，或者州或区域的简写之类的“查找”表，将它们放在一个单独的数据库中是个好主意，这样就不需要每次都备份这些表。
- 不要备份没有改变的行。如果一个表只做插入，例如记录网页页面点击的表，那么可以增加一个时间戳的列，然后只备份自上次备份后插入的行。
- 某些数据根本不需要备份。有时候这样做影响会很大——例如，如果有一个从其他数据构建的数据仓库，从技术上讲完全是冗余的，就可以仅备份构建仓库的数据，而不是数据仓库本身。即使从源数据文件重建仓库的“恢复”时间较长，这也是个好想法。相对于从全备中可能获得的快速恢复时间，避免备份可以节约更多的总的时间开销。临时数据也可以不用备份，例如保留网站会话数据的表。
- 备份所有的数据，然后发送到一个有去重特性的目的地，例如 ZFS 文件管理程序。

632 增量备份的缺点包括增加恢复复杂性，额外的风险，以及更长的恢复时间。如果可以做

注 6： Percona XtraBackup 正在开发“真正的”增量备份特性。它将能够备份变更的块，而不需要扫描每个块。

全备，考虑到简便性，我们建议尽量做全备。

不管如何，还是需要经常做全备份——建议至少一周一次。你肯定不会希望使用一个月的所有增量备份来进行恢复。即使一周也还是有很多的工作和风险的。

15.3.4 存储引擎和一致性

MySQL 对存储引擎的选择会导致备份明显更复杂。问题是，对于给定的存储引擎，如何得到一致的备份。

实际上有两类一致性需要考虑：数据一致性和文件一致性。

数据一致性

当备份时，应该考虑是否需要数据在指定时间点一致。例如，在一个电子商务数据库中，可能需要确保发货单和付款之间一致。恢复付款时如果不考虑相应的发货单，或反过来，都会导致麻烦。

如果做在线备份（从一个运行的服务器做备份），可能需要所有相关表的一致性备份。这意味着不能一次锁住一张表然后做备份——因而意味着备份可能比预想的要更有侵入性。如果使用的不是事务型存储引擎，则只能在备份时用 `LOCK TABLES` 来锁住所有要一起备份的表，备份完成后再释放锁。

InnoDB 的多版本控制功能可以帮到我们。开始一个事务，转储一组相关的表，然后提交事务。（如果使用了事务获取一致性备份，则不能用 `LOCK TABLES`，因为它会隐式地提交事务——详情参见 MySQL 手册。）只要在服务器上使用 `REPEATABLE READ` 事务隔离级别，并且没有任何 DDL，就一定会有完美的一致性，以及基于时间点的数据快照，且在备份过程中不会阻塞任何后续的工作。

尽管如此，这种方法并不能保护逻辑设计很差的应用。假如在电子商务库中插入一条付款记录，提交事务，然后在另外一个事务中插入一条发货单记录。备份过程可能在这两个操作之间开始，备份了付款记录却不包括发货单记录。这就是必须仔细设计事务以确保相关的操作放在一个组内的原因。

也可以用 *mysqldump* 来获得 InnoDB 表的一致性逻辑备份，采用 *--single-transaction* 选项可以按照我们所描述的那样工作。但是，这可能会导致一个非常长的事务，在某些负载下会导致开销大到不可接受。

633 文件一致性

每个文件的内部一致性也非常重要——例如，一条大的 UPDATE 语句执行时备份反映不出文件的状态——并且所有要备份的文件相互间也应一致。如果没有内部一致的文件，还原时可能会感到惊讶（它们可能已经损坏）。如果是在不同的时间复制相关的文件，它们彼此可能也不一致。MyISAM 的 *.MYD* 和 *.MYI* 文件就是个例子。InnoDB 如果检测到不一致或损坏，会记录错误日志乃至让服务器崩溃。

对于非事务性存储引擎，例如 MyISAM，可能的选项是锁住并刷新表。这意味着要么用 LOCK TABLES 和 FLUSH TABLES 结合的方法以使服务器将内存中的变更刷到磁盘上，要么用 FLUSH TABLES WITH READ LOCK。一旦刷新完成，就可以安全地复制 MyISAM 的原始文件。

对于 InnoDB，确保文件在磁盘上一致更困难。即使使用 FLUSH TABLES WITH READ LOCK，InnoDB 依旧在后台运行：插入缓存、日志和写线程继续将变更合并到日志和表空间文件中。这些线程设计上是异步的——在后台执行这些工作可以帮助 InnoDB 取得更高的并发性——正因为如此它们与 LOCK TABLES 无关。因此，不仅需要确保每个文件内部是一致的，还需要同时复制同一个时间点的日志和表空间文件。如果在备份时有其他线程在修改文件，或在与表空间文件不同的时间点备份日志文件，会在恢复后再次因系统损坏而告终。可以通过下面几个方法规避这个问题。

- 等待直到 InnoDB 的清除线程和插入缓冲合并线程完成。可以观察 SHOW INNODB STATUS 的输出，当没有脏缓存或挂起的写时，就可以复制文件。尽管如此，这种方法可能需要很长一段时间；因为 InnoDB 的后台线程涉及太多的干扰而不太安全。所以我们不推荐这种方法。
- 在一个类似 LVM 的系统中获取数据和日志文件一致的快照，必须让数据和日志文件在快照时相互一致；单独取它们的快照是没有意义的。在本章后续的 LVM 快照中会讨论。
- 发送一个 STOP 信号给 MySQL，做备份，然后再发送一个 CONT 信号来再次唤醒 MySQL。看起来像是一个很少推荐的方法，但如果另外一种方法是在备份过程中需要关闭服务器，则这种方法值得考虑。至少这种技术不需要在重启服务器后预热。

在复制数据文件到其他地方后，就可以释放锁以使 MySQL 服务器再次正常运行。

634 复制

从备库中备份最大的好处是可以不干扰主库，避免在主库上增加额外的负载。这是一个建立备库的好理由，即使不需要用它做负载均衡或高可用。如果钱是个问题，也可以把备份用的备库用于其他用途，例如报表服务——只要不对其做写操作，以确保备份时不

会修改数据。备库不必只用于备份的目的；只需要在下次备份时能及时跟上主库，即使有时因作为其他用途导致复制延时也没有关系。

当从备库备份时，应该保存所有关于复制进程的信息，例如备库相对于主库的位置。这对于很多情况都非常有用：克隆新的备库，重新应用二进制日志到主库上以获得指定时间点的恢复，将备库提升为主库等。如果停止备库，需要确保没有打开的临时表，因为它们可能导致不能重启备库。

故意将一个备库延时一段时间对于某些灾难场景非常有用。例如延时复制一小时，当一个不期望的语句在主库上运行后，将有一个小时的时间观察到并在从中继日志重放之前停掉复制。然后可以将备库提升为主库，重放少量相关的日志事件，跳过错误的语句。这比我们后面将要讨论的指定时间点的恢复技术可能要快很多。Percona Toolkit 中 *pt-slave-delay* 工具可以帮助实现这个方案。

> 备库可能与主库数据不完全一样。许多人认为备库是主库完全一样的副本，但以我们的经验，主库与备库数据不匹配是很常见的，并且 MySQL 没有方法检测这个问题。检测这个问题的唯一方法是使用 Percona Toolkit 中的 *pt-table-checksum* 之类的工具。
>
> 拥有一个复制的备库可能在诸如主库的硬盘烧坏时提供帮助，但却不能提供保证。复制不是备份。

15.4 管理和备份二进制日志

服务器的二进制日志是备份的最重要因素之一。它们对于基于时间点的恢复是必需的，并且通常比数据要小，所以更容易进行频繁的备份。如果有某个时间点的数据备份和所有从那时以后的二进制日志，就可以重放自从上次全备以来的二进制日志并“前滚”所有的变更。

MySQL 复制也使用二进制日志。因此备份和恢复的策略经常和复制配置相互影响。

二进制日志很“特别”。如果丢失了数据，你一定不希望同时丢失了二进制日志。为了 635
让这种情况发生的几率减少到最小，可以在不同的卷上保存数据和二进制日志。即使在 LVM 下生成二进制日志的快照，也是可以的。为了额外的安全起见，可以将它们保存在 SAN 上，或用 DRBD 复制到另外一个设备上。

经常备份二进制日志是个好主意。如果不能承受丢失超过 30 分钟数据的价值，至少要每 30 分钟就备份一次。也可以用一个配置 *--log_slave_update* 的只读备库，这样可以获得额外的安全性。备库上日志位置与主库不匹配，但找到恢复时正确的位置并不难。最后，

MySQL 5.6 版本的 *mysqlbinlog* 有一个非常方便的特性，可连接到服务器上来实时对二进制日志做镜像，比起运行一个 *mysqld* 实例要简单和轻便。它与老版本是向后兼容的。

请参考第 8 章和第 10 章中我们推荐的关于二进制日志的服务器配置。

15.4.1 二进制日志格式

二进制日志包含一系列的事件。每个事件有一个固定长度的头，其中有各种信息，例如当前时间戳和默认的数据库。可以使用 *mysqlbinlog* 工具来查看二进制日志的内容，打印出一些头信息。下面是一个输出的例子。

```
1  # at 277
2  #071030 10:47:21 server id 3  end_log_pos 369   Query   thread_id=13    exec_time=0
   error_code=0
3  SET TIMESTAMP=1193755641/*!*/;
4  insert into test(a) values(2)/*!*/;
```

第一行包含日志文件内的偏移字节值（本例中为 277）。

第二行包含如下几项。

- 事件的日期和时间，MySQL 会使用它们来产生 `SET TIMESTAMP` 语句。
- 原服务器的服务器 ID，对于防止复制之间无限循环和其他问题是非常有必要的。
- `end_log_pos`，下一个事件的偏移字节值。该值对一个多语句事务中的大部分事件是不正确的。在此类事务过程中，MySQL 的主库会复制事件到一个缓冲区，但这样做的时候它并不知道下个日志事件的位置。
- 事件类型。本例中的类型是 `Query`，但还有许多不同的类型。
- 原服务器上执行事件的线程 ID，对于审计和执行 `CONNECTION_ID()` 函数很重要。
- 636 `exec_time`，这是语句的时间戳和写入二进制日志的时间之差。不要依赖这个值，因为它可能在复制落后的备库上会有很大的偏差。
- 在原服务器上事件产生的错误代码。如果事件在一个备库上重放时导致不同的错误，那么复制将因安全预警而失败。

后续的行包含重放变更时所需的数据。用户自定义的变更和任何其他特定设置，例如当语句执行时有效的时间戳，也将会出现在这里。

> 如果使用的是 MySQL 5.1 中基于行的日志，事件将不再是 SQL。而是可读性较差的由语句对表所做变更的“镜像”。

15.4.2 安全地清除老的二进制日志

需要决定日志的过期策略以防止磁盘被二进制日志写满。日志增长多大取决于负载和日志格式（基于行的日志会导致更大的日志记录）。我们建议，如果可能，只要日志有用就尽可能保留。保留日志对于设置复制、分析服务器负载、审计和从上次全备按时间点进行恢复，都很有帮助。当决定想要保留日志多久时，应该考虑这些需求。

一个常见的设置是使用 `expire_log_days` 变量来告诉 MySQL 定期清理日志。这个变量直到 MySQL 4.1 才引入；在此之前的版本，必须手动清理二进制日志。因此，你可能看到一些用类似下面的 *cron* 项来删除老的二进制日志的建议。

```
0 0 * * * /usr/bin/find /var/log/mysql -mtime +N -name "mysql-bin.[0-9]*" | xargs rm
```

尽管这是在 MySQL 4.1 之前清除日志的唯一办法，但在新版本中不要这么做！用 *rm* 删除日志会导致 *mysql-bin.index* 状态文件与磁盘上的文件不一致，有些语句，例如 `SHOW MASTER LOGS` 可能会受到影响而悄然失败。手动修改 *mysql-bin.index* 文件也不会修复这个问题。应该用类似下面的 *cron* 命令。

```
0 0 * * * /usr/bin/mysql -e "PURGE MASTER LOGS BEFORE CURRENT_DATE - INTERVAL N DAY"
```

`expire_logs_days` 设置在服务器启动或 MySQL 切换二进制日志时生效，因此，如果二进制日志从没有增长和切换，服务器不会清除老条目。此设置是通过查看日志的修改时间而不是内容来决定哪个文件需要被清除。

15.5 备份数据 637

大多数时候，生成备份有好的也有差的方法——有时候显而易见的方法并不是好方法。一个有用的技巧是应该最大化利用网络、磁盘和 CPU 的能力以尽可能快地完成备份。这是一个需要不断去平衡的事情，必须通过实验以找到“最佳平衡点”。

15.5.1 生成逻辑备份

对于逻辑备份，首先要意识到的是它们并不是以同样方式创建的。实际上有两种类型的逻辑备份：SQL 导出和符号分隔文件。

SQL 导出

SQL 导出是很多人所熟悉的，因为它们是 *mysqldump* 默认的方式。例如，用默认选项导出一个小表将产生如下（有删减）输出。

```
$ mysqldump test t1
-- [Version and host comments]
/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
-- [More version-specific comments to save options for restore]
--
-- Table structure for table `t1`
--
DROP TABLE IF EXISTS `t1`;
CREATE TABLE `t1` (
  `a` int(11) NOT NULL,
  PRIMARY KEY  (`a`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
--
-- Dumping data for table `t1`
--
LOCK TABLES `t1` WRITE;
/*!40000 ALTER TABLE `t1` DISABLE KEYS */;
INSERT INTO `t1` VALUES (1);
/*!40000 ALTER TABLE `t1` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;
/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
-- [More option restoration]
```

导出文件包含表结构和数据，均以有效的 SQL 命令形式写出。文件以设置 MySQL 各种选项的注释开始。这些要么是为了使恢复工作更高效，要么是因为兼容性和正确性。接下来可以看到表结构，然后是数据。最后，脚本重置在导出开始时变更的选项。

导出的输出对于还原操作来说是可执行的。这很方便，但 *mysqldump* 默认选项对于生成一个巨大的备份却不是太适合（后续我们会深入介绍 *mysqldump* 的选项）。

638 *mysqldump* 不是生成 SQL 逻辑备份的唯一工具。例如，也可以用 *mydumper* 或 phpMyAdmin 工具来创建[注7]。我们想指出的是，不是某一个特定的工具有多大的问题，而是做 SQL 逻辑备份本身就有一些缺点。下面是主要问题点：

Schema 和数据存储在一起

如果想从单个文件恢复这样做会非常方便，但如果只想恢复一个表或只想恢复数据就很困难了。可以通过导出两次的方法来减缓这个问题——一次只导出数据，另外一次只导出 Schema——但还是会有下一个麻烦。

巨大的 SQL 语句

服务器分析和执行 SQL 语句的工作量非常大，所以加载数据时会非常慢。

单个巨大的文件

大部分文本编辑器不能编辑巨大的或者包含非常长的行的文件。尽管有时候可以用命令行的流编辑器——例如 *sed* 或 *grep*——来抽出需要的数据，但保持文件小型化

注 7： 请不要用 Maatkit 的 *mk-parallel-dump* 和 *mk-parallel-restore* 工具。它们并不安全。

仍然是更合适的。

逻辑备份的成本很高

比起逻辑备份这种从存储引擎中读取数据然后通过客户端 / 服务器协议发送结果集的方式，还有其他更高效的方法。

这些限制意味着 SQL 导出在表变大时可能变得不可用。不过，还有另外一个选择：导出数据到符号分隔的文件中。

符号分隔文件备份

可以使用 SQL 命令 SELECT INTO OUTFILE 以符号分隔文件格式创建数据的逻辑备份。（可以用 *mysqldump* 的 *--tab* 选项导出到符号分隔文件中）。符号分隔文件包含以 ASCII 展示的原始数据，没有 SQL、注释和列名。下面是一个导出为逗号分隔值（CVS）格式的例子，对于表格形式的数据来说这是一个很好的通用格式。

```
mysql> SELECT * INTO OUTFILE '/tmp/t1.txt'
    -> FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    -> LINES TERMINATED BY '\n'
    -> FROM test.t1;
```

比起 SQL 导出文件，符号分隔文件要更紧凑且更易于用命令行工具操作，这种方法最大的优点是备份和还原速度更快。可以和导出时使用一样的选项，用 LOAD DATA INFILE 方法加载数据到表中：

```
mysql> LOAD DATA INFILE '/tmp/t1.txt'
    -> INTO TABLE test.t1
    -> FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    -> LINES TERMINATED BY '\n';
```

下面这个非正式的测试演示了 SQL 文件和符号分隔文件在备份和还原上的速度差异。在测试中，我们对生产数据做了些修改。导出的表看起来像下面这样：

639

```
CREATE TABLE load_test (
   col1 date NOT NULL,
   col2 int NOT NULL,
   col3 smallint unsigned NOT NULL,
   col4 mediumint NOT NULL,
   col5 mediumint NOT NULL,
   col6 mediumint NOT NULL,
   col7 decimal(3,1) default NULL,
   col8 varchar(10) NOT NULL default '',
   col9 int NOT NULL,
   PRIMARY KEY  (col1, col2)
) ENGINE=InnoDB;
```

这张表有 1500 万行，占用近 700MB 的磁盘空间。表 15-1 对比了两种备份和还原方法

的性能。可以看到测试中还原时间有较大的差异。

表15-1：SQL和符号分隔导出所用的备份和恢复时间

方法	导出大小	导出时间	还原时间
SQL 导出	727 MB	102	600
符号分隔导出	669 MB	86	301

但是 SELECT INTO OUTFILE 方法也有一些限制。

- 只能备份到运行 MySQL 服务器的机器上的文件中。(可以写一个自定义的 SELECT INTO OUTFILE 程序，在读取 SELECT 结果的同时写到磁盘文件中，我们已经看到有些人采用这种方法。)
- 运行 MySQL 的系统用户必须有文件目录的写权限，因为是由 MySQL 服务器来执行文件的写入，而不是运行 SQL 命令的用户。
- 出于安全原因，不能覆盖已经存在的文件，不管文件权限如何。
- 不能直接导出到压缩文件中。
- 某些情况下很难进行正确的导出或导入，例如非标准的字符集。

640

15.5.2 文件系统快照

文件系统快照是一种非常好的在线备份方法。支持快照的文件系统能够瞬间创建用来备份的内容一致的镜像。支持快照的文件系统和设备包括 FreeBSD 的文件系统、ZFS 文件系统、GNU/Linux 的逻辑卷管理（LVM），以及许多的 SAN 系统和文件存储解决方案，例如 NetApp 存储。

不要把快照和备份相混淆。创建快照是减少必须持有锁的时间的一个简单方法；释放锁后，必须复制文件到备份中。事实上，有些时候甚至可以创建 InnoDB 快照而不需要锁定。我们将要展示两种使用 LVM 来对 InnoDB 文件系统做备份的方法，可以选择最小化锁或零锁的方案。

快照对于特别用途的备份是一个非常好的方法。一个例子是在升级过程中遇到有问题而回退的情况。可以在升级前创建一个镜像，这样如果升级有问题，只需要回滚到该镜像。可以对任何不确定和有风险的操作都这么做，例如对一个巨大的表做变更（需要多少时间是未知的）。

LVM 快照是如何工作的

LVM 使用写时复制（copy-on-write）的技术来创建快照——例如，对整个卷的某个瞬间的逻辑副本。这与数据库中的 MVCC 有点像，不同的是它只保留一个老的数据版本。

注意，我们说的不是物理副本。逻辑副本看起来好像包含了创建快照时卷中所有的数据，但实际上一开始快照是不包含数据的。相比复制数据到快照中，LVM 只是简单地标记创建快照的时间点，然后对该快照请求读数据时，实际上是从原始卷中读取的。因此，初始的复制基本上是一个瞬间就能完成的操作，不管创建快照的卷有多大。

当原始卷中某些数据有变化时，LVM 在任何变更写入之前，会复制受影响的块到快照预留的区域中。LVM 不保留数据的多个“老版本”，因此对原始卷中变更块的额外写入并不需要对快照做其他更多的工作。换句话说，对每个块只有第一次写入才会导致写时复制到预留的区域。

现在，在快照中请求这些块时，LVM 会从复制块中而不是从原始卷中读取。所以，可以继续看到快照中相同时间点的数据而不需要阻塞任何原始卷。图 15-1 描述了这个方案。

快照会在 */dev* 目录下创建一个新的逻辑卷，可以像挂载其他设备一样挂载它。

641

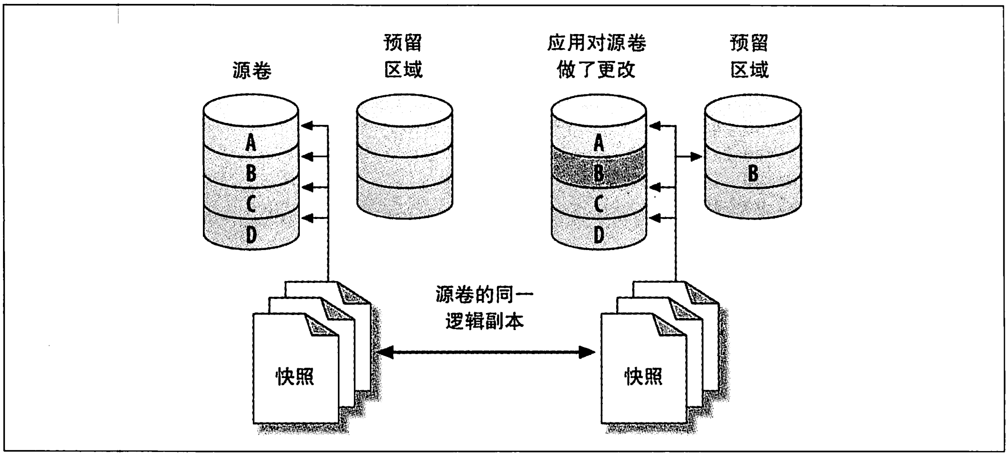


图15-1：写时复制技术如何减少单个卷快照需要的大小

理论上讲，这种技术可以对一个非常大的卷做快照，而只需要非常少的物理存储空间。但是，必须设置足够的空间，保证在快照打开时，能够保存所有期望在原始卷上更新的块。如果不预留足够的写时复制空间，当快照用完所有的空间后，设备就会变得不可用。这个影响就像拔出一个外部设备：任何从设备上读的备份工作都会因 I/O 错误而失败。

先决条件和配置

创建一个快照的消耗几乎微不足道，但还是需要确保系统配置可以让你获取在备份瞬间的所有需要的文件的一致性副本。首先，确保系统满足下面这些条件。

- 所有的 InnoDB 文件（InnoDB 的表空间文件和 InnoDB 的事务日志）必须是在单个逻辑卷（分区）。你需要绝对的时间点一致性，LVM 不能为多于一个卷做某个时间点一致的快照。（这是 LVM 的一个限制；其他一些系统没有这个问题。）
- 如果需要备份表定义，MySQL 数据目录必须在相同的逻辑卷中。如果使用另外一种方法来备份表的定义，例如只备份 Schema 到版本控制系统中，就不需要担心这个问题。
- 必须在卷组中有足够的空闲空间来创建快照。需要多少取决于负载。当配置系统时，应该留一些未分配的空间以便后面做快照。

642 LVM 有卷组的概念，它包含一个或多个逻辑卷。可以按照如下的方式查看系统中的卷组：

```
# vgs
  VG   #PV #LV #SN Attr   VSize   VFree
  vg     1   4   0 wz--n- 534.18G 249.18G
```

输出显示了一个分布在一个物理卷上的卷组，它有四个逻辑卷，大概有 250GB 空间空闲。如果需要，可用 *vgdisplay* 命令产生更详细的输出。现在让我们看一下系统上的逻辑卷：

```
# lvs
  LV    VG   Attr   LSize   Origin Snap%  Move Log Copy%
  home  vg   -wi-ao  40.00G
  mysql vg   -wi-ao 225.00G
  tmp   vg   -wi-ao  10.00G
  var   vg   -wi-ao  10.00G
```

输出显示 mysql 卷有 225GB 的空间。设备名是 /dev/vg/mysql。这仅是个名字，尽管看起来像一个文件系统路径。更加让人困惑的是，还有个符号链接从相同名字的文件链到 */dev/mapper/vg-mysql* 的设备节点，用 *ls* 和 *mount* 命令可以观察到。

```
# ls -l /dev/vg/mysql
lrwxrwxrwx 1 root root 20 Sep 19 13:08 /dev/vg/mysql -> /dev/mapper/vg-mysql
# mount | grep mysql
/dev/mapper/vg-mysql on /var/lib/mysql
```

有了这个信息，就可以创建文件系统快照了。

创建、挂载和删除 LVM 快照

一条命令就能创建快照。只需要决定快照存放的位置和分配给写时复制的空间大小即可。不要纠结于是否使用比想象中的需求更多的空间。LVM 不会马上使用完所有指定的空间，只是为后续使用预留而已。因此多预留一点空间并没有坏处，除非你必须同时为其他快照预留空间。

让我们来练习创建一个快照。我们给它 16GB 的写时复制空间，名字为 backup_mysql。

```
# lvcreate --size 16G --snapshot --name backup_mysql /dev/vg/mysql
  Logical volume "backup_mysql" created
```

这里特意命名为 backup_mysql 卷而不是 mysql_backup，是为了避免 Tab 键自动补全造成误会。这有助于避免因为 Tab 键自动补全导致突然误删除 mysql 卷组的可能。

现在让我们看看新创建的卷的状态。

```
# lvs
  LV           VG   Attr   LSize   Origin Snap%  Move Log Copy%
  backup_mysql vg   swi-a-  16.00G mysql    0.01
  home         vg   -wi-ao  40.00G
  mysql        vg   owi-ao 225.00G
  tmp          vg   -wi-ao  10.00G
  var          vg   -wi-ao  10.00G
```

643

可以注意到，快照的属性与原设备不同，而且该输出还显示了一点额外的信息：原始卷组和分配了 16GB 的写时复制空间目前已经使用了多少。备份时对此进行监控是个非常好的主意，可以知道是否会因为设备写满而备份失败。可以交互地监控设备的状态，或使用诸如 Nagios 这样的监控系统。

```
# watch 'lvs | grep backup'
```

从前面 *mount* 的输出可以看到，mysql 卷包含一个文件系统。这意味着快照也同样如此，可以像其他文件系统一样挂载。

```
# mkdir /tmp/backup
# mount /dev/mapper/vg-backup_mysql /tmp/backup
# ls -l /tmp/backup/mysql
total 5336
-rw-r----- 1 mysql mysql     0 Nov 17  2006 columns_priv.MYD
-rw-r----- 1 mysql mysql  1024 Mar 24  2007 columns_priv.MYI
-rw-r----- 1 mysql mysql  8820 Mar 24  2007 columns_priv.frm
-rw-r----- 1 mysql mysql 10512 Jul 12 10:26 db.MYD
-rw-r----- 1 mysql mysql  4096 Jul 12 10:29 db.MYI
-rw-r----- 1 mysql mysql  9494 Mar 24  2007 db.frm
... omitted ...
```

这里只是为了练习，因此我们卸载这个快照并用 *lvremove* 命令将其删除。

```
# umount /tmp/backup
# rmdir /tmp/backup
# lvremove --force /dev/vg/backup_mysql
  Logical volume "backup_mysql" successfully removed
```

用于在线备份的 LVM 快照

现在已经知道如何创建、加载和删除快照，可以使用它们来进行备份了。首先看一下如何在不停止 MySQL 服务的情况下备份 InnoDB 数据库，这里需要使用一个全局的读锁。连接 MySQL 服务器并使用一个全局读锁将表刷到磁盘上，然后获取二进制日志的位置：

```
mysql> FLUSH TABLES WITH READ LOCK; SHOW MASTER STATUS;
```

记录 SHOW MASTER STATUS 的输出，确保到 MySQL 的连接处于打开状态，以使读锁不被释放。然后获取 LVM 的快照并立刻释放该读锁，可以使用 UNLOCK TABLES 或者直接关闭连接来释放锁。最后，加载快照并复制文件到备份位置。

644 这种方法最主要的问题是，获取读锁可能需要一点时间，特别是当有许多长时间运行的查询时。当连接等待全局读锁时，所有的查询都将被阻塞，并且不可预测这会持续多久。

文件系统快照和 InnoDB

即使锁住所有的表，InnoDB 的后台线程仍会继续工作，因此，即使在创建快照时，仍然可以往文件中写入。并且，由于 InnoDB 没有执行关闭操作，如果服务器意外断电，快照中 InnoDB 的文件会和服务器意外掉电后文件的遭遇一样。

这不是什么问题，因为 InnoDB 是个 ACID 系统。任何时刻（例如快照时），每个提交的事务要么在 InnoDB 数据文件中要么在日志文件中。在还原快照后启动 MySQL 时，InnoDB 将运行恢复进程，就像服务器断过电一样。它会查找事务日志中任何提交但没有应用到数据文件中的事务然后应用，因此不会丢失任何事务。这正是要强制 InnoDB 数据文件和日志文件在一起快照的原因。

这也是在备份后需要测试的原因。启动一个 MySQL 实例，把它指向一个新备份，让 InnoDB 执行崩溃恢复过程，然后检测所有的表。通过这种方法，就不会备份损坏了却还不知道（文件可能由于任何原因损坏）。这么做的另外一个好处是，未来需要从备份中还原时会更快，因为已经在备份上运行过一遍恢复程序了。

甚至还可以在将快照复制到备份目的地之前，直接在快照上做上面的操作，但增加一点点额外开销。所以需要确保这是计划内的操作。（后面会有更多说明。）

使用 LVM 快照无锁 InnoDB 备份

无锁备份只有一点不同。区别是不需要执行 FLUSH TABLES WITH READ LOCK。这意味着不能保证 MyISAM 文件在磁盘上一致，如果只使用 InnoDB，这就不是问题。mysql 系

统数据库中依然有部分 MyISAM 表，但如果是典型的工作负载，在快照时这些表不太可能发生改变。

如果你认为 mysql 系统表可能会变更，那么可以锁住并刷新这些表。一般不会对这些表有长时间运行的查询，所以通常会很快。

```
mysql> LOCK TABLES mysql.user READ, mysql.db READ, ...;
mysql> FLUSH TABLES mysql.user, mysql.db, ...;
```

由于没有用全局读锁，因此不会从 SHOW MASTER STATUS 中获取到任何有用的信息。尽管如此，基于快照启动 MySQL（来验证备份的完整性）时，也将会在日志文件中看到像下面的内容。

```
InnoDB: Doing recovery: scanned up to log sequence number 0 40817239
InnoDB: Starting an apply batch of log records to the database...
InnoDB: Progress in percents: 3 4 5 6 ...[omitted]... 97 98 99
InnoDB: Apply batch completed
InnoDB: Last MySQL binlog file position 0 3304937, file name
/var/log/mysql/mysql-bin.000001
070928 14:08:42  InnoDB: Started; log sequence number 0 40817239
```
645

InnoDB 记录了 MySQL 已经恢复的时间点对应的二进制日志位置。这个二进制日志位置可以用来做基于时间点的恢复。

使用快照进行无锁备份的方法在 MySQL 5.0 或更新版本中有变动。这些 MySQL 版本使用 XA 来协调 InnoDB 和二进制日志。如果还原到一个与备份时 server_id 不同的服务器，服务器在准备事务阶段可能发现这是从另外一个与自己有不同 ID 的服务器来的。在这种情况下，服务器会变得困惑，恢复事务时可能会卡在 PREPARED 状态。这种情况很少发生，但是存在可能性。这也是只有经过验证才可以说备份成功的原因。有些备份也许是不能恢复的。

如果是在备库上获取快照，InnoDB 恢复时还会打印如下几行日志。

```
InnoDB: In a MySQL replica the last master binlog file
InnoDB: position 0 115, file name mysql-bin.001717
```

输出显示了 InnoDB 已经恢复的基于主库的二进制日志位置（相对于备库二进制日志位置），这对于基于备库备份或基于其他备库克隆备库来说非常有用。

规划 LVM 备份

LVM 快照备份也是有开销的。服务器写到原始卷的越多，引发的额外开销也越多。当服务器随机修改许多不同块时，磁头需要自写时复制空间来来回回寻址，并且将数据的老版本写到写时复制空间。从快照中读取也有开销，因为 LVM 需要从原始卷中读取大部

分数据。只有快照创建后修改过的数据从写时复制空间读取；因此，逻辑顺序读取快照数据实际上也可能导致磁头来回移动。

所以应该为此规划好快照。快照实际上会导致原始卷和快照都比正常的读/写性能要差——如果使用过多的写时复制空间，性能可能会差很多。这会降低MySQL服务器和复制文件进行备份的性能。我们做了基准测试，发现LVM快照的开销要远高于它本应该有的——我们发现性能最多可能会慢5倍，具体取决于负载和文件系统。在规划备份时要记得这一点。

646 规划中另外一个重要的事情是，为快照分配足够多的空间。我们一般采取下面的方法。

- 记住，LVM只需要复制每个修改块到快照一次。MySQL写一个块到原始卷中时，它会复制这个块到快照中，然后对复制的块在例外表中生成一个标记。后续对这个块的写不会产生任何到快照的复制。
- 如果只使用InnoDB，要考虑InnoDB是如何写数据的。InnoDB实际需要对数据写两遍，至少一半的InnoDB的写I/O会到双写缓冲（doublewrite buffer）、日志文件，以及其他磁盘上相对小的区域中。这部分会多次重用相同的磁盘块，因此第一次时对快照有影响，但写过一次以后就不会对快照带来写压力。
- 接下来，相对于反复修改同样的数据，需要评估有多少I/O需要写入到那些还没有复制到快照写时复制空间的块中，对评估的结果要保留足够的余量。
- 使用*vmstat*或*iostat*来收集服务器每秒写多少块的统计信息。
- 衡量（或评估）复制备份到其他地方需要多久。换言之，需要在复制期间保持LVM快照打开多长时间。

假设评估出有一半的写会导致往快照的写时复制空间的写操作，并且服务器支持10MB/s的写入。如果需要一个小时（3600s）将快照复制到另外一个服务器上，那么将需要1/2 × 10MB × 3600即18GB的快照空间。考虑到容错，还要增加一些额外的空间。

有时候当快照保持打开时，很容易计算会有多少数据发生改变。让我们看个例子。BoardReader论坛搜索引擎每个存储节点有约1TB的InnoDB表。但是，我们知道最大的开销是加载新数据。每天新增近10GB的数据，因此50GB的快照空间应该完全足够。然而这样来评估并不总是正确的。假设在某个时间点，有一个长时间运行的依次修改每个分片的`ALTER TABLE`操作，它会修改超过50GB的数据；在这个时间点，就不能做备份操作。为了避免这样的问题，可以稍后再创建快照，因为创建快照后会导致一个负载的高峰。

> ### 备份误区 2："快照就是备份"
>
> 一个快照，不论是 LVM 快照、ZFS 快照，还是 SAN 快照，都不是实际的备份，因为它不包含数据的完整副本。正因为快照是写时复制的，所以它只包含实际数据和快照发生的时间点的数据之间的差异数据。如果一个没有被修改的块在备份副本时被损坏，那就没有该块的正常副本可以用来恢复，并且备份副本时每个快照看到的都是相同的损坏的块。可以使用快照来"冻结"备份时的数据，但不要把快照当作一个备份。

快照的其他用途和替代方案

647

快照有更多的其他用途，而不仅仅用于备份。例如，之前提到，在一个有潜在危险的动作之前生成一个"检查点"会有帮助。有些系统允许将快照提升为原文件系统，这使得回滚到生成快照的时间点的数据非常简单。

文件系统快照不是取得数据瞬间副本的唯一方法。另外一个选择是 RAID 分裂：举个例子，如果有一个三磁盘的软 RAID 镜像，就可以从该 RAID 组中移出来一个磁盘单独加载。这样做没有写时复制的代价，并且需要时将此类"快照"提升为主副本的操作也很简单。不错，如果要将磁盘加回到 RAID 集合，就必须重新进行同步。当然，天下没有免费的午餐。

15.6 从备份中恢复

如何恢复数据取决于是怎么备份的。可能需要以下部分或全部步骤。

- 停止 MySQL 服务器。
- 记录服务器的配置和文件权限。
- 将数据从备份中移到 MySQL 数据目录。
- 改变配置。
- 改变文件权限。
- 以限制访问模式重启服务器，等待完成启动。
- 载入逻辑备份文件。
- 检查和重放二进制日志。
- 检测已经还原的数据。
- 以完全权限重启服务器。

我们在接下来的章节中将演示这些步骤的具体操作。我们也会对本节及本章后面几节提

及的一些特殊的备份方法和工具做一些解释。

> 如果有机会使用文件的当前版本，就不要用备份中的文件来代替。例如，如果备份包含二进制日志，并且需要重放这些日志来做基于时间点的恢复，那么不要把当前二进制日志用备份中的老的副本替代。如果有需要，可以将其重命名或移动到其他地方。

在恢复过程中，保证 MySQL 除了恢复进程外不接受其他访问，这一点往往比较重要。我们喜欢以 *--skip-networking* 和 *--socket=/tmp/mysql_recover.sock* 选项来启动 MySQL，
648 以确保它对于已经存在的应用不可访问，直到我们检测完并重新提供服务。这对于按块加载的逻辑备份的恢复来说尤其重要。

15.6.1 恢复物理备份

恢复物理备份往往非常直接——换言之，没有太多的选项。这可能是好事，也可能是坏事，具体取决于恢复的需求。一般过程是简单地复制文件到正确位置。

是否需要关闭 MySQL 取决于存储引擎。MyISAM 的文件一般相互独立，即使服务器正在运行，简单地复制每个表的 *.frm*、*.MYI* 和 *.MYD* 文件也可以正常操作。一旦有任何对此表的查询，或者其他会导致服务器访问此表的操作（例如，执行 `SHOW TABLES`），MySQL 都会立刻找到这些表。如果在复制这些文件时表是打开的，可能会有麻烦，因此操作前要么删除或重命名该表，要么使用 `LOCK TABLES` 和 `FLUSH TABLES` 来关闭它。

InnoDB 的情况有所不同。如果用传统的 InnoDB 的步骤来还原，即所有表都存储在单个表空间，就必须关闭 MySQL，复制或移动文件到正确位置上，然后重启。同样也需要 InnoDB 的事务日志文件与表空间文件匹配。如果文件不匹配——例如，替换了表空间文件但没有替换事务日志文件——InnoDB 将会拒绝启动。这也是将日志和数据文件一起备份非常关键的一个原因。

如果使用 InnoDB file-per-table 特性（`innodb_file_per_table`），InnoDB 会将每个表的数据和索引存储于一个 *.ibd* 文件中，这就像 MyISAM 的 *.MYI* 和 *.MYD* 文件合在一起。可以在服务器运行时通过复制这些文件来备份和还原单个表，但这并不像 MyISAM 中那样简单。这些文件并不完全独立于 InnoDB。每个 *.ibd* 文件都有一些内部的信息，保存着它与主（共享）表空间之间的关系。在还原这样的文件时，需要让 InnoDB 先“导入”这个文件。

这个过程有许多的限制，如果有需要可以阅读 MySQL 用户手册中关于每个表使用独立表空间中的部分。最大的限制是只能在当初备份的服务器上还原单个表。用这种配置来

备份和还原多个表不是不可能，但可能比想象的要更棘手。

> Percona Server 和 Percona XtraBackup 有一些改进，放宽了部分关于这个过程的限制，例如同一服务器的限制。

所有这些复杂度意味着还原物理备份会非常乏味，并且容易出错。一个好的值得倡导的规则是，恢复过程越难越复杂，也就越需要逻辑备份的保护。为了防止一些无法意料的 649
情况或者某些无法使用物理备份的场景，准备好逻辑备份总是值得推荐的。

还原物理备份后启动 MySQL

在启动正在恢复的 MySQL 服务器之前，还有些步骤要做。

首先，最重要且最容易忘记的事情，是在启动 MySQL 服务器之前检查服务器的配置，确保恢复的文件有正确的归属和权限。这些属性必须完全正确，否则 MySQL 可能无法启动。这些属性因系统的不同而不同，因此要仔细检查是否和之前做的记录吻合。一般都需要 *mysql* 用户和组拥有这些文件和目录，并且只有这个用户和组拥有可读 / 写权限。

建议观察 MySQL 启动时的错误日志。在 UNIX 类系统上，可以如下观察文件。

```
$ tail -f /var/log/mysql/mysql.err
```

注意错误日志的准确位置会有所不同。一旦开始监测文件，就可以启动 MySQL 服务器并监测错误。如果一切进展顺利，MySQL 启动后就有一个恢复好的数据库服务器了。

观察错误日志对于新的 MySQL 版本更为重要。老版本在 InnoDB 有错时不会启动，但新版本不管怎样都会启动，而只是让 InnoDB 失效。即使服务器看起来启动没有任何问题，也应该对每个数据库运行 `SHOW TABLE STATUS` 来再次检测错误日志。

15.6.2 还原逻辑备份

如果还原的是逻辑备份而不是物理备份，则与使用操作系统简单地复制文件到适当位置的方式不同，需要使用 MySQL 服务器本身来加载数据到表中。

在加载导出文件之前，应该先花一点时间考虑文件有多大，需要多久加载完，以及在启动之前还需要做什么事情，例如通知用户或禁掉部分应用。禁掉二进制日志也是个好主意，除非需要将还原操作复制到备库：服务器加载一个巨大的导出文件的代价很高，并且写二进制日志会增加更多的（可能没有必要的）开销。加载巨大的文件对于一些存储引擎也有影响。例如，在单个事务中加载 100GB 数据到 InnoDB 就不是个好想法，因为

650 巨大的回滚段将会导致问题。应该以可控大小的块来加载，并且逐个提交事务。有两种类型的逻辑备份，所以相应地有两种类型的还原操作。

加载 SQL 文件

如果有一个 SQL 导出文件，它将包含可执行的 SQL。需要做的就是运行这个文件。假设备份 Sakila 示例数据库和 Schema 到单个文件，下面是用来还原的常用命令。

```
$ mysql < sakila-backup.sql
```

也可以从 *mysql* 命令行客户端用 SOURCE 命令加载文件。这只是做相同事情的不同方法，不过该方法使得某些事情更简单。例如，如果你是 MySQL 管理用户，就可以关闭用客户端连接执行时的二进制记录，然后加载文件而不需要重启 MySQL 服务器。

```
mysql> SET SQL_LOG_BIN = 0;
mysql> SOURCE sakila-backup.sql;
mysql> SET SQL_LOG_BIN = 1;
```

需要注意的是，如果使用 SOURCE，当定向文件到 *mysql* 时，默认情况下，发生一个错误不会导致一批语句退出。

如果备份做过压缩，那么不要分别解压缩和加载。应该在单个操作中完成解压缩和加载。这样做会快很多。

```
$ gunzip -c sakila-backup.sql.gz | mysql
```

如果想用 SOURCE 命令加载一个压缩文件，可参考下节中关于命名管道的讨论。

如果只想恢复单个表（例如，actor 表），要怎么做呢？如果数据没有分行但有 schema 信息，那么还原数据并不难。

```
$ grep 'INSERT INTO `actor`' sakila-backup.sql | mysql sakila
```

或者，如果文件是压缩过的，那么命令如下。

```
$ gunzip -c sakila-backup.sql.gz | grep 'INSERT INTO `actor`'| mysql sakila
```

如果需要创建表并还原数据，而在单个文件中有整个数据库，则必须先编辑这个文件。这也是有一些人喜欢导出每个表到各自文件中的原因。大部分编辑器无法应付巨大的文件，尤其如果它们是压缩过的。另外，也不会想实际地编辑文件本身——只想抽取相关的行——因此可能必须做一些命令行工作。使用 *grep* 来仅抽出给定表的 INSERT 语句较简单，就像我们在前面命令中做的那样，但得到 CREATE TABLE 语句比较难。下面是抽取所需段落的 *sed* 脚本。

```
$ sed -e '/./{H;$!d;}' -e 'x;/CREATE TABLE `actor`/!d;q' sakila-backup.sql
```

我们得承认这条命令非常隐晦。如果必须以这种方式还原数据，那只能说明备份设计非常糟糕。如果有一点规划，可能就不会需要痛苦地去尝试弄清楚 *sed* 如何工作了。只需要备份每个表到各自的文件，或者可以更进一步，分别备份数据和 Schema。 651

加载符号分隔文件

如果是通过 SELECT INTO OUTFILE 导出的符号分隔文件，可以使用 LOAD DATA INFILE 通过相同的参数来加载。也可以用 *mysqlimport*，这是 LOAD DATA INFILE 的一个包装。这种方式依赖命名约定决定从哪里加载一个文件的数据。

我们希望你导出了 Schema，而不仅是数据。如果是这样，那应该是一个 SQL 导出，就可以使用上一节中描述的技术来加载。

使用 LOAD DATA INFILE 有一个非常好的优化技巧。LOAD DATA INFILE 必须直接从文本文件中读取，因此，如果是压缩文件很多人会在加载前先解压缩，这是非常慢的磁盘密集型的操作。然而，在支持 FIFO"命名管道"文件的系统如 GNU/Linux 上，对这种操作有个很好的方法。首先，创建一个命名管道并将解压缩数据流到它里面。

```
$ mkfifo /tmp/backup/default/sakila/payment.fifo
$ chmod 666 /tmp/backup/default/sakila/payment.fifo
$ gunzip -c /tmp/backup/default/sakila/payment.txt.gz
  > /tmp/backup/default/sakila/payment.fifo
```

注意我们使用了一个大于号字符（>）来重定向解压缩输出到 *payment.fifo* 文件中——而不是在不同程序之间创建匿名管道的管道符号。

管道会等待，直到其他程序打开它并从另外一端读取数据。简单一点说，MySQL 服务器可以从管道中读取解压缩后的数据，就像其他文件一样。如果可能，不要忘记禁掉二进制日志。

```
mysql> SET SQL_LOG_BIN = 0; -- Optional
    -> LOAD DATA INFILE '/tmp/backup/default/sakila/payment.fifo'
    -> INTO TABLE sakila.payment;
Query OK, 16049 rows affected (2.29 sec)
Records: 16049  Deleted: 0  Skipped: 0  Warnings: 0
```

一旦 MySQL 加载完数据，*gunzip* 就会退出，然后可以删除该命令管道。在 MySQL 命令行客户端使用 SOURCE 命令加载压缩的文件也可以使用此技术。Percona Toolkit 中的 *pt-fifo-split* 程序还可以帮助分块加载大文件，而不是在单个大事务中操作，这样效率更高。

652

你无法从这里到达那里

本书的作者之一曾将一列从 DATETIME 变为 TIMESTAMP，以节约空间并使处理过程更快，就像第 3 章中推荐的那样。结果表定义如下。

```
CREATE TABLE tbl (
   col1 timestamp NOT NULL,
   col2 timestamp NOT NULL default CURRENT_TIMESTAMP
        on update CURRENT_TIMESTAMP,
   ... more columns ...
);
```

这个表定义在 MySQL 5.0.40 版本上导致了一个语法错误，而这是创建时的版本。可以执行导出，但无法加载。这很奇怪，诸如这样无法预料的错误也是测试备份重要的原因之一。你永远不会知道什么会阻止你还原数据！

15.6.3 基于时间点的恢复

对 MySQL 做基于时间点的恢复常见的方法是还原最近一次全备份，然后从那个时间点开始重放二进制日志（有时叫“前滚恢复”）。只要有二进制日志，就可以恢复到任何希望的时间点。甚至可以不太费力地恢复单个数据库。

主要的缺点是二进制日志重放可能会是一个很慢的过程。它大体上等同于复制。如果有一个备库，并且已经测量到 SQL 线程的利用率有多高，那么对重放二进制日志会有多快就会心里有数了。例如，如果 SQL 线程约有 50% 被利用，则恢复一周二进制日志的工作可能在三到四天内完成。

一个典型场景是对有害的语句的结果做回滚操作，例如 DROP TABLE。让我们看一个简化的例子，看只有 MyISAM 表的情况下该如何做。假如是在半夜，备份任务在运行与下面所列相当的语句，复制数据库到同一服务器上的其他地方。

```
mysql> FLUSH TABLES WITH READ LOCK;
    -> server1# cp -a /var/lib/mysql/sakila /backup/sakila;
mysql> FLUSH LOGS;
    -> server1# mysql -e "SHOW MASTER STATUS" --vertical > /backup/master.info;
mysql> UNLOCK TABLES;
```

然后，假设有人在晚些时间运行下列语句。

```
mysql> USE sakila;
mysql> DROP TABLE sakila.payment;
```

为了便于说明，我们先假设可以单独地恢复这个数据库（即此库中的表不涉及跨库查询）。再假设是直到后来出问题才意识到这个有问题的语句。目标是恢复数据库中除了有问题的语句之外所有发生的事务。也就是说，其他表已经做的所有修改都必须保持，包括有问题的语句运行之后的修改。 653

这并不是很难做到。首先，停掉 MySQL 以阻止更多的修改，然后从备份中仅恢复 sakila 数据库。

```
server1# /etc/init.d/mysql stop
server1# mv /var/lib/mysql/sakila /var/lib/mysql/sakila.tmp
server1# cp -a /backup/sakila /var/lib/mysql
```

再到运行的服务器的 *my.cnf* 中添加如下配置以禁止正常的连接。

```
skip-networking
socket=/tmp/mysql_recover.sock
```

现在可以安全地启动服务器了。

```
server1# /etc/init.d/mysql start
```

下一个任务是从二进制日志中分出需要重放和忽略的语句。事发时，自半夜的备份以来，服务器只创建了一个二进制日志。我们可以用 *grep* 来检查二进制日志文件以找到问题语句。

```
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215
| grep -B3 -i 'drop table sakila.payment'
# at 352
#070919 16:11:23 server id 1  end_log_pos 429   Query   thread_id=16    exec_time=0
error_code=0
SET TIMESTAMP=1190232683/*!*/;
DROP TABLE sakila.payment/*!*/;
```

可以看到，我们想忽略的语句在日志文件中的 352 位置，下一个语句位置是 429。可以用下面的命令重放日志直到 352 位置，然后从 429 继续。

```
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215
--stop-position=352 | mysql -uroot -p
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215
--start-position=429 | mysql -uroot -p
```

接下来要做的是检测数据以确保没有问题，然后关闭服务器并撤消对 *my.cnf* 的改变，最后重启服务器。

15.6.4 更高级的恢复技术

复制和基于时间点的恢复使用的是相同的技术：服务器的二进制日志。这意味着复制在恢复时会是个非常有帮助的工具，哪怕方式不是很明显。在本节中我们将演示一些可以用到的方法。这里列出来的不是一个完全的列表，但应该可以为你根据需求设计恢复方案带来一些想法。记得编写脚本，并且对恢复过程中需要用到的所有技术进行预演。

654 用于快速恢复的延时复制

在本章的前面已经提到，如果有一个延时的备库，并且在备库执行问题语句之前就发现了问题，那么基于时间点的恢复就更快更容易了。

恢复的过程与本章前几节描述的有点不一样，但思路是相同的。停止备库，用 START SLAVE UNTIL 来重放事件直到要执行问题语句。接着，执行 SET GLOBAL SQL_SLAVE_SKIP_COUNTER=1 来跳过问题语句。如果想跳过多个事件，可以设置一个大于 1 的值（或简单地使用 CHANGE MASTER TO 来前移备库在日志中的位置）。

然后要做的就是执行 START SLAVE，让备库执行完所有的中继日志。这样就利用备库完成了基于时间点的恢复中所有冗长的工作。现在可以将备库提升为主库，整个恢复过程基本上没有中断服务。

即使没有延时的备库来加速恢复，普通的备库也有好处，至少会把主库的二进制日志复制到另外的机器上。如果主库的磁盘坏了，备库上的中继日志可能就是唯一能够获取到的最接近主库二进制日志的东西了。

使用日志服务器进行恢复

还有另外一种使用复制来做恢复的方法：设置日志服务器。我们感觉复制比 *mysqlbinlog* 更可靠，*mysqlbinlog* 可能会有一些导致异常行为的奇怪的 Bug 和不常见的情况。使用日志服务器进行恢复比 *mysqlbinlog* 更灵活更简单，不仅因为 START SLAVE UNTIL 选项，还因为那些可以采用的复制规则（例如 replicate-do-table）。使用日志服务器，相对其他的方式来说，可以做到更复杂的过滤。

例如，使用日志服务器可以轻松地恢复单个表。而用 *mysqlbinlog* 和命令行工具则要困难得多——事实上，这样做太复杂了，所以我们一般不建议进行尝试。

假设粗心的开发人员像前面的例子一样删除了同样的表，现在想恢复此误操作，但又不想让整个服务器退到昨晚的备份。下面是利用日志服务器进行恢复的步骤：

1. 将需要恢复的服务器叫作 server1。
2. 在另外一台叫做 server2 的服务器上恢复昨晚的备份。在这台服务器上运行恢复进

程，以免在恢复时犯错而导致事情更糟。

3. 按照第 10 章的做法设置日志服务器来接收 server1 的二进制日志（复制日志到另外一个服务器并设置日志服务器是个好想法，但是要格外注意。） 655
4. 改变 server2 的配置文件，增加如下内容。

```
replicate-do-table=sakila.payment
```

5. 重启 server2，然后用 CHANGE MASTER TO 来让它成为日志服务器的备库。配置它从昨晚备份的二进制日志坐标读取。这时候切记不要运行 START SLAVE。
6. 检测 server2 上的 SHOW SLAVE STATUS 的输出，验证一切正常。要三思而行！
7. 找到二进制日志中问题语句的位置，在 server2 上执行 START SLAVE UNTIL 来重放事件直到该位置。
8. 在 server2 上用 STOP SLAVE 停掉复制进程。现在应该有被删除表，因为现在从库停止在被删除之前的时间点。
9. 将所需表从 server2 复制到 server1。

只有没有任何多表的 UPDATE、DELETE 或 INSERT 语句操作这个表时，上述流程才是可行的。任何这样的多表操作语句在被记录的时候，可能是基于多个数据库的状态，而不仅仅是当前要恢复的这个数据库，所以这样恢复出来的数据可能和原始的有所不同。（只有在使用基于语句的二进制日志时才会有这个问题；如果使用的是基于行的日志，重放过程不会碰到这个错误。）

15.6.5 InnoDB 崩溃恢复

InnoDB 在每次启动时都会检测数据和日志文件，以确认是否需要执行恢复过程。而且，InnoDB 的恢复过程与我们在本章之前谈论的不是一回事。它并不是恢复备份的数据；而是根据日志文件将事务应用到数据文件，将未提交的变更从数据文件中回滚。

精确地描述 InnoDB 如何进行恢复工作，这有点太过复杂。我们要关注的焦点是当 InnoDB 有严重问题时如何实际执行恢复。

大部分情况下 InnoDB 可以很好地解决问题。除非 MySQL 有 Bug 或硬件有问题，否则不需要做任何非常规的事情，哪怕是服务器意外断电。InnoDB 会在启动时执行正常的恢复，然后就一切正常了。在日志文件中，可以看到如下信息。

```
InnoDB: Doing recovery: scanned up to log sequence number 0 40817239
InnoDB: Starting an apply batch of log records to the database...
```

InnoDB 会在日志文件中输出恢复进度的百分比信息。有些人说直到整个过程完成才能看到这些信息。耐心点，这个恢复过程是急不来的。如果心急而杀掉进程并重启，只会

导致需要更长的恢复时间。

656 如果服务器硬件有严重问题，例如内存或磁盘损坏，或遇到了 MySQL 或 InnoDB 的 Bug，可能就不得不介入，这时要么进行强制恢复，要么阻止正常恢复发生。

InnoDB 损坏的原因

InnoDB非常健壮且可靠，并且有许多的内建安全检测来防止、检测和修复损坏的数据——比其他 MySQL 存储引擎要强很多。然而，InnoDB 并不能保护自己避免一切错误。

最起码，InnoDB 依赖于无缓存的 I/O 调用和 `fsync()` 调用，直到数据完全地写入到物理介质上才会返回。如果硬件不能保证写入的持久化，InnoDB 也就不能保证数据的持久，崩溃就有可能导致数据损坏。

很多 InnoDB 损坏问题都是与硬件有关的（例如，因电力问题或内存损坏而导致损坏页的写入）。然而，在我们的经验中，错误配置的硬件是更多的问题之源。常见的错误配置包括打开了不包含电池备份单元的 RAID 卡的回写缓存，或打开了硬盘驱动器本身的回写缓存。这些错误将会导致控制器或驱动器“撒谎”，在数据实际上只写入到回写缓存上而不是磁盘上时，却说 `fsync()` 已经完成。换句话说，硬件没有提供保持 InnoDB 数据安全的保证。

有时候机器默认就会这样配置，因为这样做可以得到更好的性能——对于某些场景确实很好，但是对事务数据服务来说却是个大问题。

如果在网络附加存储（NAS）上运行 InnoDB，也可能会遇到损坏，因为对 NAS 设备来说完成 `fsync()` 只是意味着设备接收到了数据。如果 InnoDB 崩溃，数据是安全的，但如果是 NAS 设备崩溃就不一定了。

严重的损坏会使 InnoDB 或 MySQL 崩溃，而不那么严重的损坏则可能只是由于日志文件未真正同步到磁盘而丢掉了某些事务。

如何恢复损坏的 InnoDB 数据

InnoDB 损坏有三种主要类型，它们对数据恢复有着不同程度的要求。

二级索引损坏

一般可以用 `OPTIMIZE TABLE` 来修复损坏的二级索引；此外，也可以用 `SELECT INTO OUTFILE`，删除和重建表，然后 `LOAD DATA INFILE` 的方法。（也可以将表改为使用 MyISAM 再改回来。）这些过程都是通过构建一个新表重建受影响的索引，来修复损坏的索引数据。

聚簇索引损坏 657

如果是聚簇索引损坏，也许只能使用 innodb_force_recovery 选项来导出表（关于这点后续会讲更多）。有时导出过程会让 InnoDB 崩溃；如果出现这样的情况，或许需要跳过导致崩溃的损坏页以导出其他的记录。聚簇索引的损坏比二级索引要更难修复，因为它会影响数据行本身，但在多数场合下仍然只需要修复受影响的表。

损坏系统结构

系统结构包括 InnoDB 事务日志、表空间的撤销日志（undo log）区域和数据字典。这种损坏可能需要做整个数据库的导出和还原，因为 InnoDB 内部绝大部分的工作都可能受到影响。

一般可以修复损坏的二级索引而不丢失数据。然而，另外两种情形经常会引起数据的丢失。如果已经有备份，那最好还是从备份中还原，而不是试着从损坏的文件里去提取数据。

如果必须从损坏的文件里提取数据，那一般过程是先尝试让 InnoDB 运行起来，然后使用 SELECT INTO OUTFILE 导出数据。如果服务器已经崩溃，并且每次启动 InnoDB 都会崩溃，那么可以配置 InnoDB 停止常规恢复和后台进程的运行。这样也许可以启动服务器，然后在缺少或不做完整性检查的情况下做逻辑备份。

innodb_force_recovery 参数控制着 InnoDB 在启动和常规操作时要做哪一种类型的操作。通常情况下这个值是 0，可以增大到 6。MySQL 使用手册里记录了每个数值究竟会产生什么行为；在此我们不会重复这段信息，但是要告诉你：在有点危险的前提下，可以把这个数值调高到 4。使用这个设置时，若有数据页损坏，将会丢失一些数据；如果将数值设得更高，可能会从损坏的页里提取到坏掉的数据，或者增加执行 SELECT INTO OUTFILES 时崩溃的风险。换句话说，这个值直到 4 都对数据没有损害，但可能丧失修复问题的机会；而到 5 和 6 会更主动地修复问题，但损害数据的风险也会很大。

当把 innodb_force_recovery 设为大于 0 的某个值时，InnoDB 基本上是只读的，但是仍然可以创建和删除表。这可以阻止进一步的损坏，InnoDB 会放松一些常规检查，以便在发现坏数据时不会特意崩溃。在常规操作中，这样做是有安全保障的，但是在恢复时，最好还是避免这样做。如果需要执行 InnoDB 强制恢复，有个好主意是配置 MySQL，使它在操作完成之前不接受常规的连接请求。

如果 InnoDB 的数据损坏到了根本不能启动 MySQL 的程度，还可以使用 Percona 出品的 658
InnoDB Recovery Toolkit 从表空间的数据文件里直接抽取数据。这个工具由本书的几个作者开发，可以从 *http://www.percona.com/software* 免费获取。Percona Server 还有允许服务器在某些表损坏时仍能运行的选项，而不是像 MySQL 那样在单个表损坏页被检测出时就默认强制崩溃。

15.7 备份和恢复工具

有各种各样的好的和不是那么好的备份工具。我们喜欢对 LVM 使用 *mylvmbackup* 做快照备份，使用 Percona Xtrabackup（开源）或 MySQL Enterprise Backup（收费）做 InnoDB 热备份。不建议对大数据量使用 *mysqldump*，因为它对服务器有影响，并且漫长的还原时间不可预知。

有一些备份工具已经出现多年了，不幸的是有些已经过时。最明显的例子是 Maatkit 的 *mk-parallel-dump*，它从没有正确运行，甚至被重新设计过好几次还是不行。另外一个工具是 *mysqlhotcopy*，它适合于古老的 MyISAM 表。大部分场景下这两个工具都无法让人相信数据是安全的，它们会使人误以为备份了数据实际上却非如此。例如，当使用 InnoDB 的 `innodb_file_per_table` 时，*mysqlhotcopy* 会复制 *.ibd* 文件，这会使一些人误以为 InnoDB 的数据已经备份完成。在某些场景下，这两个工具都对服务器有一些负面影响。

如果你在 2008 或 2009 年时在看 MySQL 的路线图，可能听说过 MySQL 在线备份。这是一个可以用 SQL 命令来开始备份和还原的特性。它原本是规划在 MySQL 5.2 版本中，后来重新安排在了 MySQL 6.0 中，再后来，据我们所知被永久取消了。

15.7.1 MySQL Enterprise Backup

这个工具之前叫做 InnoDB Hot Backup 或 *ibbackup*，是从 Oracle 购买的 MySQL Enterprise 中的一部分。使用此工具备份不需要停止 MySQL，也不需要设置锁或中断正常的数据库活动（但是会对服务器造成一些额外的负载）。它支持类似压缩备份、增量备份和到其他服务器的流备份的特性。这是 MySQL“官方”的备份工具。

15.7.2 Percona XtraBackup

Percona XtraBackup 与 MySQL Enterprise Backup 在很多方面都非常类似，但它是开源并且免费的。除了核心备份工具外，还有一个用 Perl 写的封装脚本，可以提供更多高级功能。它支持类似流、增量、压缩和多线程（并行）备份操作。也有许多特别的功能，用以降低在高负载的系统上备份的影响。

659 Percona XtraBackup 的工作方式是在后台线程不断追踪 InnoDB 日志文件尾部，然后复制 InnoDB 数据文件。这是个轻量级侵入过程，依靠特别的检测机制确保复制的数据是一致的。当所有的数据文件被复制完，日志复制线程就结束了。结果是在不同的时间点的所有数据的副本。然后可以使用 InnoDB 崩溃恢复代码应用事务日志，以达到所有数据文件一致的状态。这一步叫作准备过程。一旦准备好，备份就会完全一致，并且包含

文件复制过程最后时间点已经提交的事务。一切都在 MySQL 外部完成，因此不需要以任何方式连接或访问 MySQL。

包装脚本包含通过复制备份到原位置的方式进行恢复的能力。还有 Lachlan Mulcahy 的 XtraBack Manager 项目，功能更多，详情参见 *http://code.google.com/p/xtrabackup-manager/*。

15.7.3 mylvmbackup

Lenz Grimmer 的 *mylvmbackup*（*http://lenz.homelinux.org/mylvmbackup/*）是一个 Perl 脚本，它通过 LVM 快照帮助 MySQL 自动备份。此工具首先获取全局读锁，创建快照，释放锁。然后通过 *tar* 压缩数据并移除快照。它通过备份时的时间戳命名压缩包。它还有几个高级选项，但总的来说，这是一个执行 LVM 备份的非常简单明了的工具。

15.7.4 Zmanda Recovery Manager

适用于 MySQL 的 Zmanda Recovery Manager，或 ZRM（*http://www.zmanda.com*），有免费（GPL）和商业两种版本。企业版提供基于网页图形接口的控制台，用来配置、备份、验证、恢复、报告和调度。开源的版本包含了所有核心功能，但缺少一些额外的特性，例如基于网页的控制台。

正如其名，ZRM 实际上是一个备份和恢复管理器，而并非单一工具。它封装了自有的基于标准工具和技术，例如 *mysqldump*、LVM 快照和 Percona XtraBackup 等之上的功能。它将许多冗长的备份和恢复工作进行了自动化。

15.7.5 mydumper

几名 MySQL 现在和之前的工程师利用他们多年的经验创建了 *mydumper*，用来替代 *mysqldump*。这是一个多线程（并发）的备份和还原 MySQL 和 Drizzle 的工具集，有许多很好的特性。大概有许多人会发现多线程备份和还原的速度是这个工具最吸引人的特色。尽管我们知道有些人在生产环境中使用，但我们还没有在任何产品中使用的经验。 660
可以在 *http://www.mydumper.org* 找到更多信息。

15.7.6 mysqldump

大部分人在使用这个与 MySQL 一起发行的程序，因此，尽管它有缺点，但创建数据和 Schema 的逻辑备份最常见的选择还是 *mysqldump*。这是一个通用工具，可以用于许多的任务，例如在服务器间复制表。

```
$ mysqldump --host=server1 test t1 | mysql --host=server2 test
```

我们在本章中展示了几个用 *mysqldump* 创建逻辑备份的例子。该工具默认会输出包含创建表和填充数据的所有需要的命令；也有选项可以控制输出视图、存储代码和触发器。下面有一些典型的例子。

- 对服务器上所有的内容创建逻辑备份到单个文件中，每个库中所有的表在相同逻辑时间点备份：

  ```
  $ mysqldump --all-databases > dump.sql
  ```

- 创建只包含 Sakila 示例数据库的逻辑备份：

  ```
  $ mysqldump --databases sakila > dump.sql
  ```

- 创建只包含 `sakila.actor` 表的逻辑备份：

  ```
  $ mysqldump sakila actor > dump.sql
  ```

可以使用 *--result-file* 选项来指定输出文件，这可以帮助防止在 Windows 上发生换行符转换：

```
$ mysqldump sakila actor --result-file=dump.sql
```

mysqldump 的默认选项对于大多数备份目的来说并不够好。多半要显式地指定某些选项以改变输出。下面是一些我们经常使用的选项，可以让 *mysqldump* 更加高效，输出更容易使用。

--opt

启用一组优化选项，包括关闭缓冲区（它会使服务器耗尽内存），导出数据时把更多的数据写在更少的 SQL 语句里，以便在加载的时候更有效率，以及做其他一些有用的事情。更多细节可以阅读帮助文件。如果关闭了这组选项，*mysqldump* 会在把表写到磁盘之前，把它们都导出到内存里，这对于大型的表而言是不切实际的。

--allow-keywords, --quote-names

使用户在导出和恢复表时，可以使用保留字作为表的名字。

661 *--complete-insert*

使用户能在不完全相同列的表之间移动数据。

--tz-utc

使用户能在具有不同时区的服务器之间移动数据。

--lock-all-tables

使用 `FLUSH TABLE WITH READ LOCK` 来获取全局一致的备份。

--tab

用 SELECT INTO OUTFILE 导出文件。

--skip-extended-insert

使每一行数据都有自己的 INSERT 语句。必要时这可以用于有选择地还原某些行。它的代价是文件更大，导入到 MySQL 时开销会更大。因此，要确保只有在需要时才启用它。

如果在 *mysqldump* 上使用 *--databases* 或 *--all-databases* 选项，那么最终导出的数据在每个数据库中都一致，因为 *mysqldump* 会在同一时间锁定并导出一个数据库里的所有表。然而，来自不同数据库的各个表就未必是相互一致的。使用 *--lock-all-tables* 选项可以解决这个问题。

对于 InnoDB 备份，应该增加 *--single-transaction* 选项，这会使用 InnoDB 的 MVCC 特性在单个时间点创建一个一致的备份，而不需要使用 LOCK TABLES 锁定所有表。如果增加 *--master-data* 选项，备份还会包括在备份时服务器的二进制日志文件位置，这对基于时间点的恢复和设置复制非常有帮助。然而也要知道，获得日志位置时需要使用 FLUSH TABLES WITH READ LOCK 冻结服务器。

15.8 备份脚本化

为备份写一些脚本是标准做法。展示一个示例程序，其中必定有很多辅助内容，这只会增加篇幅，在这里我们更愿意列举一些典型的备份脚本功能，展示一些 Perl 脚本的代码片断。你可以把这些当作可重用的代码块，在创建自己的脚本时可以直接组合起来使用。下面将大致按照使用顺序来展示。

安全检测

安全检测可以让自己和同事的生活更简单点——打开严格的错误检测，并且使用英文变量名。

```
use strict;
use warnings FATAL => 'all';
use English qw(-no_match_vars);
```

如果是在 Bash 下使用脚本，还可以做更严格的变量检测。下面的设置会让替换中有未定义的变量或程序出错退出时产生一个错误。

```
set -u;
set -e;
```

662 *命令行参数*

增加命令行选项处理最好的方法是用标准库，它已包含在 Perl 标准安装中。

```
use Getopt::Long;
Getopt::Long::Configure('no_ignore_case', 'bundling');
GetOptions( .... );
```

连接 MySQL

标准的 Perl DBI 库几乎无所不在，提供了许多强大和灵活的功能。使用详情请参阅 Perldoc（可从 *http://search.cpna.org* 在线获取）。可以像下面这样使用 DBI 来连接 MySQL。

```
use DBI;
$dbh = DBI->connect(
        'DBI:mysql:;host=localhost', 'user', 'p4ssw0rd', {RaiseError => 1 });
```

对于编写命令行脚本，请阅读标准 *mysql* 程序的 *--help* 参数的输出文本。它有许多选项可更友好地支持脚本。例如，在 Bash 中遍历数据库列表如下。

```
mysql -ss -e 'SHOW DATABASES' | while read DB; do
   echo "${DB}"
done
```

停止和启动 MySQL

停止和启动 MySQL 最好的方法是使用操作系统推荐的方法，例如运行 */etc/init.d/mysql init* 脚本或通过服务控制（在 Windows 下）。然而这并不是唯一的方法。可以从 Perl 中用一个已存在的数据库连接来关闭数据库。

```
$dbh->func("shutdown", 'admin');
```

当这个命令完成时不要太指望 MySQL 已经被关闭——它可能正在关闭的过程中。也可以通过命令行来停掉 MySQL。

```
$ mysqladmin shutdown
```

获取数据库和表的列表

每个备份脚本都会查询 MySQL 以获取数据库和表的列表。要注意那些实际上并不是数据库的条目，例如一些日志系统中的 *lost+found* 文件夹和 INFORMATION_SCHEMA。也要确保脚本已经准备好应付视图，同时也要知道 SHOW TABLE STATUS 在 InnoDB 中有大量数据时可能耗时很长。

```
mysql> SHOW DATABASES;
mysql> SHOW /*!50002 FULL*/ TABLES FROM <database>;
mysql> SHOW TABLE STATUS FROM <database>;
```

对表加锁、刷新并解锁 663

如果需要对一个或多个表加锁并且/或刷新，要么按名字锁住所需的表，要么使用全局锁锁住所有的表。

```
mysql> LOCK TABLES <database.table> READ [, ...];
mysql> FLUSH TABLES;
mysql> FLUSH TABLES <database.table> [, ...];
mysql> FLUSH TABLES WITH READ LOCK;
mysql> UNLOCK TABLES;
```

在获取所有的表并锁住它们时要格外注意竞争条件。期间可能会有新表创建，或有表被删除或重命名。如果一个表一个表地锁住然后备份，将无法得到一致性的备份。

刷新二进制日志

让服务器开始一个新的二进制日志非常简单（一般在锁住表后但在备份前做这个操作）：

```
mysql> FLUSH LOGS;
```

这样做使得恢复和增量备份更简单，因为不需要考虑从一个日志文件中间开始操作。此操作会有一些副作用，比如刷新和重新打开错误日志，也可能销毁老的日志条目，因此，注意不要扔掉需要用到的数据。

获取二进制日志位置

脚本应该获取并记录主库和备库的状态——即使服务器仅是个主库或备库。

```
mysql> SHOW MASTER STATUS\G
mysql> SHOW SLAVE STATUS\G
```

执行这两条语句并忽略错误，以使脚本可以获取到所有可能的信息。

导出数据

最好的选择是使用 *mysqldump*、*mydumper* 或 SELECT INTO OUTFILE。

复制数据

可以使用本章中演示的任何一个方法。

这些都是构造备份脚本的基础。比较困难的部分是将管理和恢复任务脚本化。如果想获得实现的灵感，可以看看 ZRM 的源码。

15.9 总结 664

每个人都知道需要备份，但并不是每个人都意识到需要的是可恢复的备份。有许多方法可以规划能满足恢复需求的备份。为了避免这个问题，我们建议明确并记录恢复点目标和恢复时间目标，并且在选择备份系统时将其作为参考。

在日常基础上做恢复测试以确保备份可以正常工作也很重要。设置 *mysqldump* 并让它在每天晚上运行是很简单的，但很多时候不会意识到数据随着时间已经增长到可能需要几天或几周才能再次导入的地步。最糟糕的是当你真正需要恢复的时候，才发现原来需要这么长时间。毫不夸张地说，一个在几个小时内完成的备份可能需要几周时间来恢复，具体取决于硬件、Schema、索引和数据。

不要掉进备库就是备份的陷阱。备库对生成备份是一个干涉较少的源，但它不是备份本身。对于 RAID 卷、SAN 和文件系统快照，也同样如此。确保备份可以通过 DROP TABLE 测试（或“遭受黑客攻击”的测试），也要能通过数据中心失败的测试。如果是基于备库生成备份，确保使用 *pt-table-checksum* 验证复制的完整性。

我们最喜欢的两种备份方式，一种是从文件系统或者 SAN 快照中直接复制数据文件，一种是使用 Percona XtraBackup 做热备份。这两种方法都可以无侵入地实现二进制的原始数据备份，这样的备份可以通过启动 *mysqld* 实例检查所有的表进行验证。有时候甚至可以一石二鸟：可以在开发或者预发环境每天将备份进行还原来执行恢复测试，然后再将数据导出为逻辑备份。我们也建议备份二进制日志，并且尽可能久地保留多份备份的数据和二进制文件。这样即使最近的备份无法使用了，还可以使用较老的备份来执行恢复或者创建新的备库。

除了提到的许多开源工具，也有很多很好的商业备份工具，其中最重要的是 MySQL Enterprise Backup。对包括在 GUI SQL 编辑器、服务器管理工具和类似工具中的“备份”工具要特别小心。同样地，有一些出品“一招吃遍天下”的备份工具的公司，对于它们宣称的支持 MySQL 的“MySQL 备份插件”也要特别小心。我们需要的是主要为 MySQL 设计的优秀备份工具，而不是一个支持上百个其他数据库并恰巧支持 MySQL 的工具。有许多备份工具的供应者并不知道或明白诸如 FLUSH TABLES WITH READ LOCK 操作对数据库的影响。在我们看来，使用这种 SQL 命令的方案应该自动退出“热”备份的行列。如果只使用 InnoDB 表，就更加不需要这类工具。

第 16 章 MySQL用户工具

MySQL 服务器发行包中并没有包含针对许多常用任务的工具，例如监控服务器或比较不同服务器间数据的工具。幸运的是，Oracle 的商业版提供了一些扩展工具，并且 MySQL 活跃的开源社区和第三方公司也提供了一系列的工具，降低了自己“重复发明轮子”的需要。

16.1 接口工具

接口工具可以帮助运行查询，创建表和用户，以及执行其他日常任务等。本节将简单介绍一些用于此用途的最流行的工具。一般可以用 SQL 查询或命令做所有这些或其中大部分的工作——我们这里讨论的工具只是更为方便，可帮助避免错误和加快工作。

MySQL Workbench

MySQL Workbench 是一个一站式的工具，可以完成例如管理服务器、写查询、开发存储过程，以及 Schema 设计图相关的工作。可以通过一个插件接口来编写自己的工具并集成到这个工作平台上，有一些 Python 脚本和库就使用了这个插件接口。MySQL Workbench 有社区版和商业版两个版本，商业版只是增加了其他的一些高级特性。免费版对于大部分需要早已足够了。在 *http://www.mysql.com/products/workbench/* 可以学到更多相关的内容。

SQLyog

SQLyog 是 MySQL 最流行的可视化工具之一，有许多很好的特性。它与 MySQL Workbench 是同级别的工具，但两个工具都有一些对方没有的特性。SQLyog 只能在微软的 Windows 下使用，拥有全部特性的版本需要付费，但有限制功能的免费版本。关于 SQLyog 的更多信息可以参考 *http://www.webyog.com*。

666 *phpMyAdmin*

phpMyAdmin 是一个流行的管理工具，运行在 Web 服务器上，并且提供基于浏览器的 MySQL 服务器访问接口。尽管基于浏览器的访问有时很好，但 phpMyAdmin 是个大而复杂的工具，曾被指责有许多安全问题。对此要格外小心。我们建议不要安装在任何可以从互联网访问的地方。更多信息请参考 *http://sourceforge.net/projects/phpmyadmin/*。

Adminer

Adminer 是个基于浏览器的安全的轻量级管理工具，它与 phpMyAdmin 同类。其开发者将其定位为 phpMyAdmin 的更好的替代品。尽管它看起来更安全，但我们仍建议安装在任何可公开访问的地方时要谨慎。更多详情可参考 *http://www.adminer.org*。

16.2 命令行工具集

MySQL 包含了一些命令行工具集，例如 *mysqladmin* 和 *mysqlcheck*。这些在 MySQL 手册上都有提及和记录。MySQL 社区同样创建了大量高质量的工具包，并有很好的文档支撑这些实用工具集。

Percona Toolkit

Percona Toolkit 是 MySQL 管理员必备的工具包。它源自 Baron 早期的工具包 Maatkit 和 Aspersa，很多人认为这两个工具应该是正式的 MySQL 部署必须强制要求使用的。Percona Toolkit 包括许多针对类似日志分析、复制完整性检测、数据同步、模式和索引分析、查询建议和数据归档目的的工具。如果刚开始接触 MySQL，我们建议首先学习这些关键的工具：*pt-mysql-summary*、*pt-table-checksum*、*pt-table-sync* 和 *pt-query-digest*。更多信息可参考 *http://www.percona.com/software/*。

Maatkit and Aspersa

这两个工具约从 2006 年以某种形式出现，两者都被认为是 MySQL 用户的基本工具。它们现在已经并入 Percona Toolkit。

The openark kit

Shlomi Noach 的 openark kit（*http://code.openark.org/forge/openark-kit*）包含了可以用来做一系列管理任务的 Python 脚本。

MySQL Workbench 工具集

MySQL Workbench 工具集中的某些工具可以作为单独的 Python 脚本使用。可参考 *https://launchpad.net/mysql-utilities*。

667 除了这些工具外，还有其他一系列没有太正式包装和维护的工具。许多杰出的 MySQL 社区成员时不时地贡献工具，其中大多数托管在他们自己的网站或 MySQL Forge（*http://*

forge.mysql.com）上。可以通过不时地查看 Planet MySQL 博客聚合器获取大量的信息（*http://planet.mysql.com*），但不幸的是这些工具没有一个集中的目录。

16.3 SQL 实用集

服务器本身也内置有一系列免费的附加组件和实用集可以使用；其中一些确实相当强大。

common_schema

Shlomi Noach 的 common_schema 项目（*http://code.openark.org/forge/common_schema*）是一套针对服务器脚本化和管理的强大的代码和视图。common_schema 对于 MySQL 好比 jQuery 对于 JavaScript。

mysql-sr-lib

Giuseppe Maxia 为 MySQL 创建了一个存储过程的代码库，可以在 *http://www.nongnu.org/mysql-sr-lib/* 找到。

MySQL UDF 仓库

Roland Bouman 建立了一个 MySQL 自定义函数的收藏馆，可以在 *http://www.mysqludf.org* 获取。

MySQL Forge

在 MySQL Forge 上（*http://forge.mysql.com*），可以找到上百个社区贡献的程序、脚本、代码片断、实用集和技巧及陷阱。

16.4 监测工具

以我们的经验来看，大多数 MySQL 商店需要提供两种类型的监测工具：健康监测工具——检测到异常时告警——和为趋势、诊断、问题排查、容量规划等记录指标的工具。大多数系统仅在这些任务中的一个方面做得很好，而不能两者兼顾。更不幸的是，有十几种工具可选，使得评估和选择一款适合的工具非常耗时。

许多监控系统不是专门为 MySQL 服务器设计。它们是通用系统，用于周期性地检测许多种类型的资源，从机器到路由再到软件（例如 MySQL）。它们一般有某些类型的插件架构，经常会伴随有一些 MySQL 插件。

一般会在专用服务器上安装监控系统来监测其他服务器。如果是监控重要的系统，它很快会变成架构中至关重要的一部分，因此可能需要采取额外的步骤，例如做监控系统本身的灾备。

668 16.4.1 开源的监控工具

下面是一些最受欢迎的开源集成监控系统。

Nagios

Nagios（*http://www.nagios.org*）也许是开源世界中最流行的问题检测和告警系统。它周期性检测监控的服务器并将结果与默认或自定义的阈值相比较。如果结果超出了限制，Nagios会执行某个程序并且（或）把问题的告警发给某些人。Nagios的通信和告警系统可以将告警发给不同的联系人，改变告警，或根据一天中的时间和其他条件将其发送到不同的位置，并且对计划内的宕机可以特殊处理。Nagios同样理解服务之间的依赖，因此，如果是因为中间的路由层宕机或者主机本身宕机导致MySQL实例不可用，Nagios不会发送告警来烦你。

Nagios能将任何一个可执行文件以插件形式运行，只要给予其正确参数就可得到正确输出。因此，Nagios插件在多种语言中都存在，例如shell、Perl、Python、Ruby和其他脚本语言。就算找不到一个能真正满足你需求的插件，自己创建一个也很简单。一个插件只需要接收标准的参数，以一个合适的状态退出，然后选择性地打印Nagios捕获的输出。

然而，Nagios也有一些严重的缺点。即使你很了解它，也仍然难以维护。它将所有配置保存在文件而不是数据库中。文件有一个特别容易出错的语法，当系统增长和发展时，修改配置文件就很费事。Nagios可扩展性并不好；你可以很容易地写出监控插件，但这也就是你能够做的一切。最后，它的图形化、趋势化和可视化能力都有限。Nagios将一些性能和其他数据存储到MySQL服务器中，一般从中生成图形，但并不像其他一些系统那么灵活。因为不同"政见"的原因，使得上面所有的问题继续变得更糟。因为或真实、或臆测的涉及代码、参与者的问题，Nagios至少分化出了两个分支。两个分支的名字分别是Opsview（*http://www.opsview.com*）和Icinga（*http://www.icinga.org*）。它们比Nagios更受到人们的亲睐。

有一些专门介绍Nagios的书籍；我们倾向于Wolfgang Barth的*Nagios System and Network Monitoring*（No Starch出版公司）。

Zabbix

Zabbix是一个同时支持监控和指标收集的完整系统。例如，它将所有配置和其他数据存储到数据库而不是配置文件中。它存储了比Nagios更多的数据类型，因而可以得到更好的趋势和历史报表。其网络画图和可视能力也比Nagios更强，配置更简单，更灵活，且更具可扩展性。可参考*http://www.zabbix.com*获取更多信息。

669 *Zenoss*

Zenoss是用Python写的，拥有一个基于浏览器的用户界面，使用了Ajax，这使它更快和更高效。它可以自动发现网络上的资源，并将监控、告警、趋势、绘图和记

录历史数据整合到了一个统一的工具中。Zenoss 默认使用 SNMP 来从远程服务器上收集数据，但也可以使用 SSH，并且支持 Nagios 插件。更多信息请参考 *http://www.zenoss.com*。

Hyperic HQ

Hyperic HQ 是一个基于 Java 的监控系统，比起同级别的其他大部分系统，它更称得上是企业级监控。像 Zenoss 一样，它可以自动发现网络上的资源和支持 Nagios 插件，但它的逻辑组织和架构不同，有点"笨重"。更多信息可参考 *http://www.hyperic.com*。

OpenNMS

OpenNMS 也是用 Java 开发，有一个活跃的开发社区。它拥有常规的特性，例如监控和告警，但同样也增加了绘图和趋势功能。它的目标是高性能、可扩展、自动化和灵活。像 Hyperic 一样，它也致力于为大型和关键系统做企业级监控。更多信息请参考 *http://www.opennms.org*。

Groundwork Open Source

Groundwork Open Source 用一个可移植的接口把 Nagios 和其他几个工具整合到了一个系统中。对于这个工具最好的描述是：如果你是 Nagios、Cacti 和其他几个工具方面的专家，并且花了许多时间将它们整合一起，那很可能你是在闭门造车。更多信息可参考 *http://www.gwos.com*。

相比于集所有功能于一身的系统，还有一系列软件专注于收集指标和画图以及可视化，而不是进行性能监控检查。他们中有很多是建立在 RRDTool(*http://www.rrdtool.org*)之上，存储时序数据到轮询数据库（RRD）文件中。RRD 文件自动聚集输入数据，对没有预期传送的输入值进行插值，并有强大的绘图工具可以生成漂亮有特色的图。有很多基于 RRDTool 的系统，下面是其中最受欢迎的几个。

MRTG

Multi Router Traffic Grapher 或称 MRTG（*http://oss.oetiker.ch/mrtg/*），是典型的基于 RRDTool 的系统。最初是为记录网络流量而设计的，但同样可以扩展到用于对其他指标进行记录和绘图。

Cacti

Cacti（*http://www.cacti.net*）可能是最流行的基于 RRDTool 的系统。它采用 PHP 网页来与 RRDTool 进行交互，并使用 MySQL 数据库来定义服务器、插件、图像等。因为是模板驱动，故而可以定义模板然后应用到系统上。Baron 为 MySQL 和其他系统写了一组非常流行的模板；更多信息请参考 *http://code.google.com/p/mysql-cacti-templates/*。这些也已经被移植到 Munin、OpenNMS 和 Zabbix。 670

Ganglia

Ganglia（*http://ganglia.sourceforge.net*）与 Cacti 类似，但是为监控集群和网格系统而设计，所以可以汇总查看许多服务器的数据，如果需要也可以细分查看单台服务器的详细数据。

Munin

Munin（*http://munin.projects.linpro.no*）收集数据并存入 RRDTool 中，然后以几个不同级别的粒度生成数据图。它从配置中生成静态 HTML 文件，因此可以很容易地浏览和查看趋势。定义一个图形较容易；只需要创建一个插件脚本，其命令行帮助输出有一些 Munin 可以识别的特别语法的画图指令。

基于 RRDTool 的系统有些限制，例如不能用标准查询语言来查询存储的数据，不能永久保留数据，存在某些数据不能轻松地使用简单计数器和标准数值表示的问题，需要预先定义指标和图形等。理想情况下，我们需要的监控系统可以接受任何发送给它的指标，而不需要预先进行定义，并且后续可以绘制任意需要的图形，也不需要预先进行定义。可能我们所看到的最接近的系统是 Graphite（*http://graphite.wikidot.com*）。

这些系统都可以用来对 MySQL 收集、记录和绘制数据图表并且生成报表，有着不同程度的灵活性，目标也稍微有些不同。但它们都缺乏真正可以在问题出现时及时告警的灵活性。

我们提到的大多数系统的主要问题是，它们明显是由那些因为现有系统不能满足他们所有需求的人设计的，因此他们又重复设计了另一个无法完全满足其他人的所有需求的系统。大部分这样的系统都有一些基础的限制，例如使用一个奇怪的数据模型存储内部数据，而导致在很多场合都无法很好地工作。在很多时候，这都令人沮丧，使用这些系统都像是把一个圆形的钉子钉到了一个方形的洞里面。

16.4.2 商业监控系统

尽管我们知道许多 MySQL 用户热衷使用开源工具，但也有许多人愿意为合适的软件买单，只要这些软件可以让工作更好地完成，为他们节省时间，减少烦恼。下面是一些可以利用的商业选件。

MySQL Enterprise Monitor

MySQL Enterprise Monitor 包含在 Oracle 的 MySQL 支持服务中。它将监控、指标
671 和画图、咨询服务和查询分析等特性整合到了一个工具中。通过在服务器上使用
agent 来监测状态计数器（也包含操作系统的关键指标）。它能以两种方式抓取查询：
通过 MySQL 代理（MySQL Proxy），或使用合适的 MySQL 连接器，例如 Java 的

Connector/J 或 PHP 的 MySQLi。尽管是为监控 MySQL 而设计的，但某种程度上也可以进行扩展。同样，这个工具也无法监控基础架构中所有的服务器和所有的服务。更多信息请参考 *http://www.mysql.com/products/enterprise/monitor.html*。

MONyog

MONyog（*http://www.webyog.com*）是一个运行在桌面上的基于浏览器且无 agent 的监控系统。它会启动一个 HTTP 服务器，然后就可以通过浏览器来使用此工具。

New Relic

New Relic（*http://newrelic.com*）是一个托管式的软件即服务（Saas）的应用性能管理系统，它可以分析整个应用的性能，从应用代码（采用 Ruby，PHP，Java 和其他语言）到运行在浏览器上的 JavaScript，到数据库的 SQL 调用，甚至是服务器的磁盘空间，CPU 利用率和其它指标。

Circonus

Circonus（*https://circonus.com*）是一个源于 OmniTI 的托管式的软件即服务（SaaS）的指标和告警系统。通过 agent 从一个或多个服务器上收集指标并转发到 Circonus，然后就可以通过一个基于浏览器的仪表盘来查看。

Monitis

Monitis（*http://monitis.com*）是另外一个云托管式的软件即服务（SaaS）的监控系统。它被设计成监控"一切"，这意味着它有点普遍性。它有一个入门级的免费版 Monitor.us（*http://mon.itor.us*），也有支持 MySQL 的插件。

Splunk

Splunk（*http://www.splunk.com*）是一个日志聚集器和搜索引擎，可以帮助获得环境中所有机器生成的数据并进行运营分析。

Pingdom

Pingdom（*http://www.pingdom.com*）从世界的多个位置来监控网站的可用性和性能。实际上有许多像 Pingdom 一样的服务，我们并不需要特别推荐某一个这样的服务，但是我们确实建议使用一些外部的监控服务，以便让你在网站不可用时能够及时得到通知。很多类似的服务远不止 Ping 或获取网页。

还有许多其他的商业监控工具——我们可以凭印象列举出十几个或更多。对所有监控系统而言，要注意的一点是它们对服务器的影响。有些工具相当直白，因为它们由一些没有实际的大型高负载 MySQL 系统经验的公司设计。例如，我们不止一次通过禁止每分 672
钟对所有的数据库执行 一次 `SHOW TABLE STATUS` 的监控功能来解决突发事件。（这个命令在高 I/O 限制的系统上特别有破坏性。）频繁查询 `INFORMATION_SCHEMA` 表的工具也会导致负面影响。

16.4.3 Innotop 的命令行监控

有一些基于命令行的监测工具，它们大部分在某种方面模拟了 UNIX 中的 *top* 工具。其中最精致和最胜任的是 *innotop*（*http://code.google.com/p/innotop/*），我们将详细探讨。此外，还有几个其他的工具，例如 *mtop*（*http://mtop.sourceforge.net*）、*mytop*（*http://jeremy.zawodny.com/mysql/mytop/*）和一些基于网页的 *mytop* 克隆版本。

尽管 *mytop* 是 MySQL 上最原始的 *top* 克隆，但 *innotop* 比 *mytop* 拥有更多功能，这也是我们看重 *innotop* 的原因。

本书的作者之一 Balon Schwartz 编写了 *innotop*。它展示了服务器正在发生事情的实时更新视图。别去理会它的名称，实际上它不仅仅用于监控 InnoDB，还可以监控 MySQL 任何其他的方面。它也能同时监控多个 MySQL 实例，极具可配置性和可扩展性。

它的功能特性包括以下这些：

- 事务列表可以显示 InnoDB 当前的全部事务。
- 查询列表可以显示当前正在运行的查询。
- 可以显示当前锁和锁等待的列表。
- 以相对值显示服务器状态和变量的汇总信息。
- 有多种模式可用来显示 InnoDB 内部信息，例如缓冲区、死锁、外键错误、I/O 活动情况、行操作、信号量，以及其他更多的内容。
- 复制监控，将主服务器和从服务器的状态显示在一起。
- 显示任意服务器变量的模式。
- 服务器组可以更方便地组织多台服务器。
- 在命令行脚本下可以使用非交互式模式。

innotop 的安装很容易，可以从操作系统的软件仓库安装，也可以从 *http://code.google.com/p/innotop/* 下载到本地，然后解压缩，运行标准的 `make install` 安装过程。

```
perl Makefile.PL
make install
```

一旦安装完成，就可以在命令行里执行 *innotop*，然后它会引导你完成连接到 MySQL 实例的过程。引导过程会读取 *~/.my.cnf* 选项文件，这样，除了输入服务器的主机名和按几
673 次 Enter 键之外，什么都不用做。连接完成以后，就处在 T（InnoDB Transaction）模式了，这时，应该可看到 InnoDB 事务列表，如图 16-1 所示。

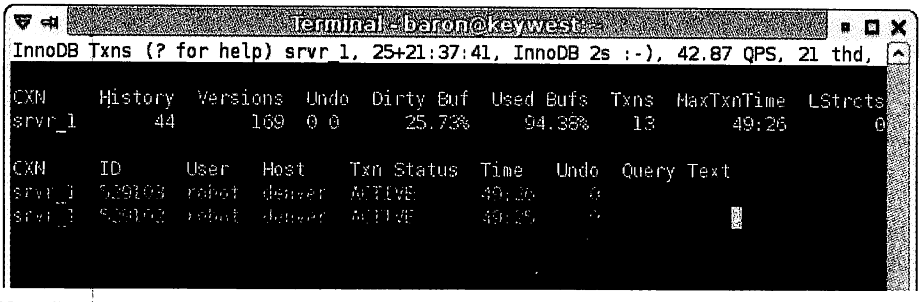


图16-1：处在T（InnoDB Transaction）模式的innotop

默认情况下，*innotop* 采用过滤器来减少零乱的信息（对于显示的所有信息，都可以定义自己的过滤器或者定制内部的过滤器）。在图 16-1 里，大多数事务都已经被过滤掉了，只显示出了当前活动的事务。可以按 i 键禁掉过滤，让数量众多的事务信息填满整个屏幕。

innotop 在这个模式下会显示头部信息和主线程列表。头部信息里显示一些 InnoDB 的总体信息，例如，历史清单的长度、还未清除的 InnoDB 事务数目、缓冲池中脏缓冲所占的百分比等。

你要按的第一个键应该是问号（?），以查看帮助信息。虽然在屏幕上显示出的帮助内容会根据当前模式的不同而不同，但是每一个活动的键都总是会显示出来，因此能看到所有可执行的动作。图 16-2 显示的是 T 模式下的帮助信息。 603

在这里不会详细讲解所有的模式，但还是可以从帮助信息里看出，*innotop* 有许许多多的功能特性。

这里唯一要提及的是一些基本的自定义功能，告诉你如何监控想要监控的信息。*innotop* 的强大功能之一就是能够解释用户定义的表达式，例如 Uptime/Questions 是生成每秒钟的查询指标。它会显示自服务器启动以来和／或自上次采样之后递增累加的结果值。

这使得往显示表格里添加自己的列方便很多。例如，在 Q（Query List）模式下，头部信息能显示出服务器的一些总体信息。让我们看看怎么将它修改一下，使它能显示出索引键缓存有多满。启动 *innotop*，按下 Q 键进入 Q 模式。这时的操作结果看起来像图 16-3 一样。

这个屏幕截图只截取了一部分，因为在这个练习里，我们对查询列表没有兴趣；我们只关心头部信息。

674

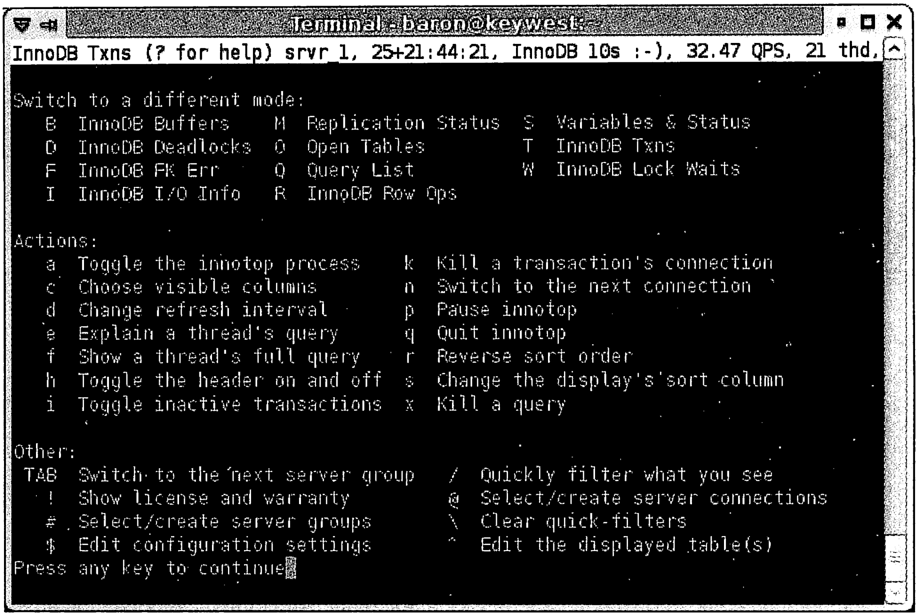


图16-2：innotop 帮助信息

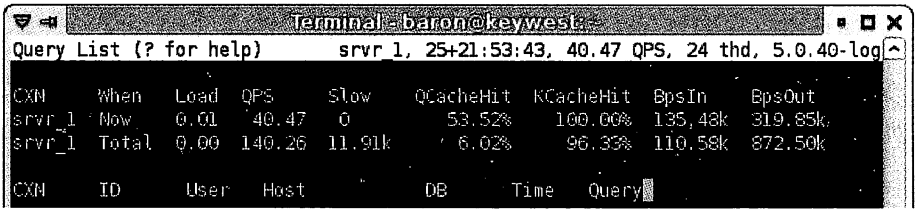


图16-3：Q模式（查询列表）下的innotop

头部显示了“当前”统计（统计自从上次 *innotop* 用服务器上的新数据刷新后的累计增量）和“总计”统计（统计自 MySQL 服务器启动以来所有的活动，这个实例中是 25 天前）。头部的每一列都是来自 SHOW STATUS 和 SHOW VARIABLES 相对应的变量值。图 16-3 中显示的头部是内建的，但也很容易增加自定义的。需要做的只是增加一列到头部“表”。按 ^ 键来打开表编辑器，然后在提示符后输入 q_header 来编辑头部表（图 16-4）。由于内置有 Tab 键自动补齐功能，因此可以敲入 q 然后按 Tab 键来补充完成整个词。

675

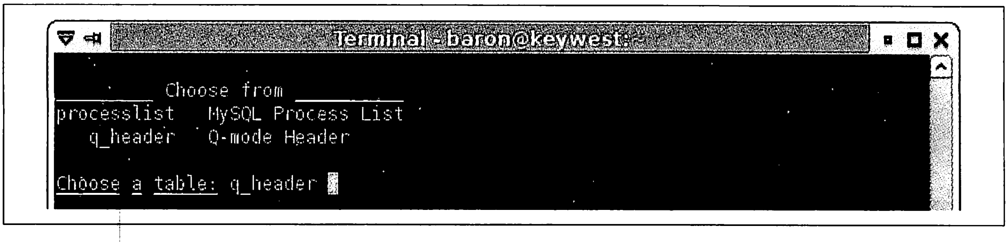


图16-4：增加一个头部（开始）

在此之后，你将会看到Q模式头部的表定义（图16-5）。该表定义显示了表的列。第一列被选中。我们可以移动选项，重新排序和编辑列，还可做其他的很多事情（按？键可以看到一个完整的列表），但我们只打算创建一个新列。按n键然后输入列名（图16-6）。

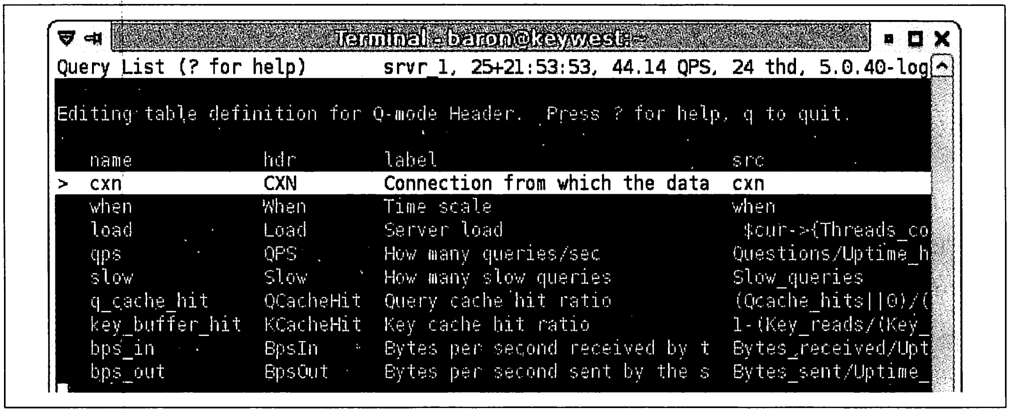


图16-5：增加头部（选择）

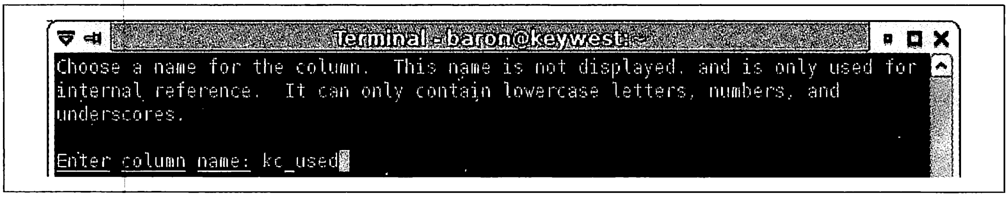


图16-6：增加头部（命名列）

接着，输入列的头部，它将在列的顶部显示（图16-7）。最后，选择列源。这是一个 676
innotop 内部编译为函数的表达式。你可以使用 SHOW VARIABLES 和 SHOW STATUS 中对应变量的名字，就像是方程中的变量一样。我们使用了一些括号和Perl式“或”默认值以

防止被零除，除此而外这个等式相当直白。我们同样可以使用 *innotop* 中的 `percent()` 转换来以百分比形式格式化结果列；更多信息请参考 *innotop* 的文档。图 16-8 显示了这个表达式。

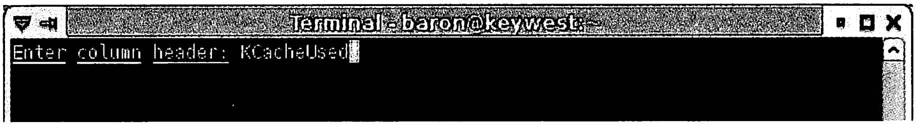


图16-7：增加头部（列的文本）

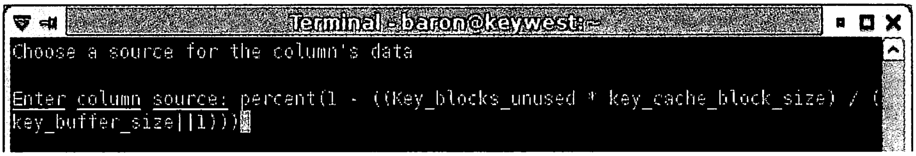


图16-8：增加头部（要计算的表达式）

按 Enter 键，你将会和之前一样看到表的定义，但是在底部有了新增加的列。按几次 + 键将它往列表上方移，挨着 `key_buffer_hit` 列，然后按 q 键退出表编辑器。瞧，新的列嵌在 KCacheHit 和 BpsIn 之间（图 16-9）。可以通过定制 *innotop* 很容易地监控想要的信息。如果它真的不能满足你的需求，甚至还可以编写对应的插件。更多文档见 *http://code.google.com/p/innotop/*。

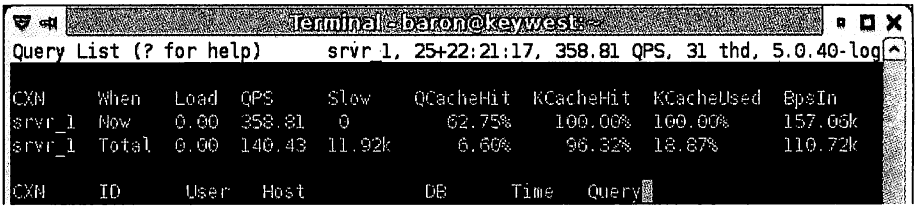


图16-9：增加头（结果）

677 16.5 总结

好的工具对管理 MySQL 至关重要。推荐使用一些已经可用、广泛测试过、流行的工具，例如 Percona Toolkit（旧名 Maatkit）。当接触新的服务器时，实践中我们首先要做的是

运行 *pt-summary* 和 *pt-mysql-summary*。如果在一台服务器上工作，可能需要在另外一个终端下运行 *innotop* 来观察它以及任何相关的服务器。

监控工具是另外一个更复杂的话题，这是由于它们对于管理非常重要。如果你是一名开源倡导者，想使用开源的监控系统，或许可以尝试 Nagios 结合带 Baron 的 Cacti 模板的 Cacti，或者尝试 Zabbix，前提是作不介意复杂的接口。如果想要监控 MySQL 的商业工具，MySQL Enterprise Monitor 可以胜任，我们知道有很多用户使用得很好。如果想监控整个环境和其中所有软硬件信息，你可能需要自己去做一些调查——这个话题超出了本书讨论的范围。

附录 A <680>

MySQL分支与变种

在第 1 章中，我们已经讨论过 MySQL 的历史：先被 Sun 公司收购，然后再被 Oracle 公司收购，以及它怎样成功度过了这些管理职务上的变动。这个故事有太多事情可讲。MySQL 不再只可从 Oracle 获取。在两次转让的过程中，出现了好几个 MySQL 变种。尽管大部分人都只愿意要 Oracle“官方”版本的 MySQL，但这些变种非常重要，并且对所有 MySQL 用户产生了一个很大的改变——甚至是那些从来没有打算使用它们的用户。

在过去几年里，出现了几个 MySQL 变种，但到目前为止主要有三个久经考验的主流变种：Percona Server，MariaDB 和 Drizzle。它们都有活跃的用户社区和某种程度上的商业支持，均由独立的服务供应商支持。

作为 Percona Server 的创建者，我们有一定的倾向性，但我们认为这个附录相当客观，因为我们对所有的 MySQL 变种都提供服务、支持、咨询、培训和工程部署。我们还邀请了 Drizzle 的创建者 Brian Aker 和 MariaDB 的创建者 Monty Widenius 来参与到这个附录的编写，因此这并非我们一家之言。

Percona Server

Percona Server（*http://www.percona.com/software/*）因我们致力于解决客户问题而衍生。在本书的第二版中，我们提到了我们为改进 MySQL 服务器的日志方法所做的补丁。那正是 Percona Server 的起源。当遇到用其他方法都不能解决的问题时，我们就去修改服务器源码。

Percona Server 有三个主要的目标。 <680>

透明

增加允许用户更紧密地查看服务器内部信息和行为的方法。包含的特性有类似 SHOW STATUS 中的计数器，INFORMATION_SCHEMA 中的表，以及慢查询日志中特别增加的详细信息。

性能

Percona Server 包含许多性能和可扩展性方面的改进。原始性能非常重要，但 Percona Server 还加强了性能的可预测性和稳定性。其中主要集中于 InnoDB。

操作灵活性

Percona Server 包含许多移除限制的特性。尽管某些限制看起来不起眼，但这些限制可能会使操作人员和系统管理员在让 MySQL 作为架构的一部分而可靠并稳定运行时，感到非常困难。

Percona Server 是个与 MySQL 向后兼容的替代品，它尽可能不改变 SQL 语法、客户端/服务器协议和磁盘上的文件格式。[注1] 任何运行在 MySQL 上的都可以运行在 Percona Server 上而不需要修改。切换到 Percona Server 只需要关闭 MySQL 和启动 Percona Server，不需要导出和重新导入数据。切换回去也不麻烦，而这一点实际上非常重要：许多问题是通过临时切换解决的，使用增强的方法来诊断，然后切回到标准 MySQL。

我们只对标准 MySQL 中需要并且可以产生显著好处的地方做改进。我们相信大部分用户坚持使用由 Oracle 发行的 MySQL 官方版本可能是最好的选择，并且努力与原版保持尽可能地相同。

Percona Server 包括 Percona XtraDB 存储引擎，即改进版本的 InnoDB。这同样是个向后兼容的替代品。例如，如果创建一个使用 InnoDB 存储引擎的表，Percona Server 能自动识别并用 Percona XtraDB 替代之。Percona XtraDB 同样包括在 MariaDB 内。

Percona Server 的一些改进已经包括在 MySQL 的 Oracle 版本中，许多其他改进也只是稍作修改而重新实现。结果，Percona Server 变成了许多特性的“抢鲜”版，这些特性随后将在标准 MySQL 中出现。Percona Server 在 5.1 和 5.5 版本中的许多改进可能要在 MySQL 5.6 中重新实现。

681

MariaDB

在 Sun 收购 MySQL 后，Monty Widenius，这位 MySQL 的创建者，因不认同 MySQL 开发流程而离开 Sun。他成立了 Monty 程序公司，创立了 MariaDB，以培养一个“开放的开发环境以鼓励外部的参与”。MariaDB 的目标是社区开发，Bug 修复和许多的新特性——

注 1： 曾有一些对文件格式的改变，但已经默认被禁掉，如果想要，还可以使其生效。

特别是与社区开发的特性相集成。再引用 Monty 的一句话，[注2]“MariaDB 的远景是面向用户和客户驱动，以及更多社区的补丁和插件。”

MariaDB有什么不同呢？与Percona Server相比，它包括了更多对服务器的扩展。(Percona Server 的大部分改变是在于 Percona XtraDB 存储引擎，而不是服务器层。）例如，有许多是对查询优化和复制的改变。它使用 Aria 存储引擎取代了 MyISAM 来存储内部临时表（被用于复杂的查询，例如 `DISTINCT` 或子查询）。Aria 最初叫 Maria，在不确定的 Sun 时代是打算替代 InnoDB 的。它是 MyISAM 的崩溃安全的版本。

除 Percona XtraDB 和 Aria 外，MariaDB 还包括许多社区的存储引擎，例如 SphinxSE 和 PBXT。

MariaDB 是原版 MySQL 的超集，因此已有的系统不需要任何修改就可以运行，就像 Percona Server 一样。然而，MariaDB 对有些场景可以更好地胜任，例如复杂的子查询或多表关联。它同样有 MyISAM 分段的键缓存，这样特性使得 MyISAM 在现代的硬件上可以更好地扩展。

也许 MariaDB 的最佳版本是 5.3，在写作此书时它还是候选发布状态。这个版本包含许多查询优化方面的工作——可能是最近十年对 MySQL 最大的优化。它增加了查询执行计划，例如哈希联合，并且修复了我们之前在本书中指出的 MySQL 的一些缺陷，例如动态列、基于角色的访问控制和微秒级时间戳的支持。

关于 MariaDB 更多的改进可参考 *http://www.askmonty.org* 上的文档或 *http://askmonty.org/blog/the-2-year-old-mariadb/* 和 *http://kb.askmonty.org/en/what-is-mariadb-53* 中的变更总结。

Drizzle

682

Drizzle 是真正的 MySQL 分支，而非只是个变种或增强版本。它并不与 MySQL 兼容，尽管区分上还并不是大相径庭。在许多场合并不能简单地将 MySQL 后端替换为 Drizzle，因为它对 SQL 语法修改太大了。

Drizzle 创建于 2008 年，致力于更好地服务 MySQL 用户。其创建目标是更好地满足网页应用的核心功能。它是个很了不起的改进，与 MySQL 相比更简单，选择更少；例如，它只使用 `utf8` 作为存储字符集，并且只有一种类型的 `BLOB`。它主要针对 64 位硬件编译，且支持 IPv6 网络。

注 2： 这句话见于 *http://askmonty.org/blog/the-2-year-old-mariadb* 和 *http://kb.askmonty.org/en/what-is-mariadb-53*。

Drizzle 数据库服务器的一个关键目标是消除 MySQL 上异常和遗留的行为，例如声明了 NOT NULL 列但发现数据库中莫名其妙地存储了 NULL。你可以在 MySQL 上找到的差劲的实现或难使用的特性已经被删除，例如触发器、查询缓存和 INSERT ON DUPLICATE KEY UPDATE。

在代码层，Drizzle 构建于一个精简内核和插件的微核心架构之上。服务器的核心比起 MySQL 已经精简许多。几乎任何东西都是插件——甚至类似 SLEEP() 的函数。这使得 Drizzle 在源码级非常简单并非常高效。

Drizzle 使用了诸如 Boost 的标准开源库，并遵从代码、构建架构和 API 方面的标准。它对类似复制等特意使用了 Google 协议缓冲公开消息格式，并且使用修改版的 InnoDB 作为标准存储引擎。

Drizzle 团队很早就开始着手做服务器的基准测试，用基于业界标准的 1024 个线程基准来评估高并发的性能。并发越大性能增加越高，对性能改进非常大。

Drizzle 是一个社区开发的项目，在开源社区比 MySQL 更吸引人。该服务器的许可证是纯 GPL 的，没有双重的许可证。然而，MySQL 客户端—服务器协议依靠一个基于 BSD 许可证的新客户端库完成，而这对于开发商业系统是最重要的一个方面。这意味着你可以通过用 Drizzle 的客户端库来连到 MySQL 的方式构建一个专属应用，并且不需要为 MySQL 客户端库购买商业许可证或将软件基于 GPL 发布。MySQL 的 *libmysql* 客户端库是众多公司为 MySQL 购买商业许可证的最主要的原因之一，没有这个链接到 *libmysql* 的商业许可证，这些公司就要被迫在 GPL 下发行软件。而这不再必要，因为现在公司可以使用 Drizzle 的库来替代。

683 但据我们了解，Drizzle 虽已在某些产品环境下部署但还没有广泛应用。Drizzle 项目的理念是抛弃向后兼容的束缚，而这意味着相对于迁移一个已有的应用而言，它更适合新的应用。

其他 MySQL 变种

现在，或曾经，有许多 MySQL 服务器的变种。许多大型公司，例如 Google、Facebook 和 eBay，都维护着这一服务器的修改版，以完全匹配其需求和部署场景。许多源码已可公开获取；也许最著名的例子就是 Facebook 和 Google 做的 MySQL 补丁。

另外还有几个分支或再发行，例如 OurDelta、DorsalSource，还有只存在了很短一段时间的 Henrik Ingo 的一个发行。

最后，许多人没有意识到当他们从 GNU/Linux 发行包软件库中安装 MySQL 时，其实获

取的是一个修改后的服务器版本——在某些场合下，有大量的修改。Red Hat 和 Debian（相应的 Fedora 和 Ubuntu）都发行了非标准版本的 MySQL，Gentoo 以及实际上任何其他 GNU/Linux 发行也都如此。与其他我们提及的变种相比，这些发行并没有指出对服务器源码做了哪些修改，因为它们保留了 MySQL 的名字。

过去我们遇到过许多关于 MySQL 修改版的问题。这是我们倾向于倡导使用 Oracle 版本的 MySQL 的一个原因，除非有很强有力的理由来使用其他版本。

总结

MySQL 分支和变种很少有大量的代码被采用到 MySQL 代码的主干树上，但却很大程度上影响了 MySQL 开发的方向和节奏。在某些情况下，它们提供了一个出众的替代选择。

应该使用分支代替 Oracle 官方的 MySQL？我们并不认为这通常有必要。如何选择一般基于理解（从来没有完全的精确）或商业原因，例如与 Oracle 有一个企业范围的关系。通常有两类人倾向不使用官方版本的服务器。

- 遇到只有改源码才能解决的特别问题的人。 684
- 不信任 Oracle 对 MySQL 的管理并且视分支为真正的开源进而快乐的人。

为什么选择某个分支？我们总结如下。如果你想与官方 MySQL 版本尽量保持紧密，并且想获取更好的性能、指导和有用的特性，那就选择 Percona Server。如果你觉得 MariaDB 对服务器的大量修改更优，或想要一个在社区内更广泛发行的存储引擎，就选择 MariaDB。如果你想要一个轻量精简版的数据库服务器并且并不介意是否与 MySQL 兼容，或想让自己对数据库的改进更容易，那就追随 Drizzle。

讲到 Percona，一般认为所有的提供商都有许多关于官方版本 MySQL 的经验，然而很自然地 Percona 对于 Percona Server 最有经验，而 Monty 公司最熟悉 MariaDB。当寻求官方发行的 MySQL 的 Bug 修复时这会有影响。只有 Oracle 能保证一个 Bug 在官方的 MySQL 发行中被修复；其他供应商可以提供修复但没有把它们加入到官方发行中的权力。这回答了为什么选择某个分支：有些人选择分支就是因为其服务供应商提供的 MySQL 版本完全可控，并且可以方便地修复和改进。

MySQL服务器状态

你可以通过查看 MySQL 的状态来回答许多关于 MySQL 的问题。MySQL 以多种方式来暴露服务器内部信息。最新的是 MySQL 5.5 中的 PERFORMANCE_SCHEMA 库，而标准的 INFORMATION_SCHEMA 库从 MySQL 5.0 就已开始存在，此外实际上一直存在一系列的 SHOW 命令。有些通过 SHOW 命令获取的信息并不在 INFORMATION_SCHEMA 中存在。

对你的挑战是，问题到底是什么，如何获取需要的信息，如何解释它。尽管 MySQL 允许你查看许多服务器内部发生的信息，但使用这些信息并不总是简单的。理解它需要耐心、经验，并要准备好参阅 MySQL 用户手册。同样，好的工具也非常有用。

这个附录大部分是参考材料，但也有许多关于服务器内部功能的信息，特别是在关于 InnoDB 的小节中。

系统变量

MySQL 通过 SHOW VARIABLES SQL 命令显露了许多系统变量，你可以在表达式中使用这些变量，或在命令行中通过 *mysqladmin variables* 试验。自 MySQL 5.1 起，可以通过访问 INFORMATION_SCHEMA 库中的表来获取这些信息。

这些变量反映了一系列配置信息，例如服务器的默认存储引擎（storage_engine）、可用的时区、连接的排序规则（collation）和启动参数。我们在第 8 章中已经讨论过如何设置和使用它们。

686 SHOW STATUS

SHOW STATUS命令会显示每个服务器变量的名字和值。和上面讲的服务器参数不一样，状态变量是只读的。可以在MySQL客户端里运行SHOW STATUS或在命令行里运行*mysqladmin extended-status*来查看这些变量。如果使用SQL命令，可以使用LIKE或WHERE来限制结果。可以用LIKE对变量名做标准模式匹配。命令将返回一个结果表，但不能对它排序，与另外一个表做联合操作，或像对MySQL表一样做一些事情。在MySQL 5.1或更新版本中，可以直接从INFORMATION_SCHEMA.GLOBAL_STATUS和INFORMATION_SCHEMA.SESSION_STATUS表中查询值。

> 我们使用“状态变量”这个术语来指从SHOW STATUS中得到的值，术语“系统变量”则指服务器配置变量。

SHOW STATUS的行为自MySQL 5.0后有了非常大的改变，但是如果你没有足够细致地观察，可能不会注意到。5.0之前的版本只有全局变量，5.1及以后的版本中，有的变量是全局的，有的变量是连接级别的。因此，SHOW STATUS混杂了全局和会话变量。其中许多变量有双重域：既是全局变量，也是会话变量，它们拥有相同的名字。现在SHOW STATUS默认也显示会话变量，因此，如果你习惯于使用SHOW STATUS来查看全局变量，则需要改为运行SHOW GLOBAL STATUS查看。[注1]

有上百个状态变量。大部分要么是计数器，要么包含某些状态指标的当前值。每次MySQL做一些事情都会导致计数器的增长，比如开始初始化一个全表扫描（Select_scan）。度量值，例如打开的到服务器连接数量（Threads_connected），可能增长和减少。有时候几个变量貌似指向相同的事情，例如Connections（尝试连接到服务器的连接数量）和Threads_connected；在本例下，变量是关联的，但类似的名字并不总是隐含某种关系。

变量采用无符号整型存储。它们在32位编译系统上用4个字节（byte），而在64位环境上用8个字节，并且当达到最大值后会重新从0开始。如果你增量地监测这些变量，可能需要观察并修正这个绕回处理；你也要意识到如果服务器已经运行很长一段时间，可能会有比预期更小的值，这是因为这些变量值已经被重置为零。（在64位编译系统上基本不会出现。）

687 如果想对服务器的工作负载有一个大体上的了解，可以将相关的一组变量放在一起查看和对比——例如，一起查看所有的Select_*变量，或所有的Handler_*变量。如果使用*innotop*，在Command Summary模式下查看更简单，但也可以通过类似*mysqladmin*

注1： 有个问题需要说明：如果在一个新版服务器上使用老版的*mysqladmin*，它不会使用SHOW GLOBAL STATUS，因此仍将不能显示“正确的”信息。

*extended -r -i60 | grep Handler_* 的命令手动完成。以下是在一个我们检测的服务器上 *innotop* 对 Select_* 变量的显示。

```
_________________ Command Summary __________________
Name                     Value   Pct     Last Incr  Pct
Select_scan             756582  59.89%           2 100.00%
Select_range            497675  39.40%           0   0.00%
Select_full_join          7847   0.62%           0   0.00%
Select_full_range_join    1159   0.09%           0   0.00%
Select_range_check           1   0.00%           0   0.00%
```

前两列是自服务器启动后的值，最后两列是自上次刷新后的值（在本例中是 10s 之前）。百分比是与打印输出中显示的总值相比较，而不是与所有查询的总值相比。

查看一组变量的当前值、上一次查询的值，以及它们之间的差值，可以使用 Percona Toolkit 中的 *pt-mext* 工具，或 Shlomi Noach 写的简洁的查询。[注2]

```
SELECT STRAIGHT_JOIN
   LOWER(gs0.VARIABLE_NAME) AS variable_name,
   gs0.VARIABLE_VALUE AS value_0,
   gs1.VARIABLE_VALUE AS value_1,
   (gs1.VARIABLE_VALUE - gs0.VARIABLE_VALUE) AS diff,
   (gs1.VARIABLE_VALUE - gs0.VARIABLE_VALUE) / 10 AS per_sec,
   (gs1.VARIABLE_VALUE - gs0.VARIABLE_VALUE) * 60 / 10 AS per_min
FROM (
     SELECT VARIABLE_NAME, VARIABLE_VALUE
     FROM INFORMATION_SCHEMA.GLOBAL_STATUS
     UNION ALL
     SELECT '', SLEEP(10) FROM DUAL
   ) AS gs0
   JOIN INFORMATION_SCHEMA.GLOBAL_STATUS gs1 USING (VARIABLE_NAME)
WHERE gs1.VARIABLE_VALUE <> gs0.VARIABLE_VALUE;
+-----------------------+---------+---------+------+---------+---------+
| variable_name         | value_0 | value_1 | diff | per_sec | per_min |
+-----------------------+---------+---------+------+---------+---------+
| handler_read_rnd_next | 2366    | 2953    |  587 |    58.7 |    3522 |
| handler_write         | 2340    | 3218    |  878 |    87.8 |    5268 |
| open_files            | 22      | 20      |   -2 |    -0.2 |     -12 |
| select_full_join      | 2       | 3       |    1 |     0.1 |       6 |
| select_scan           | 7       | 9       |    2 |     0.2 |      12 |
+-----------------------+---------+---------+------+---------+---------+
```

最有帮助的是查看整个过程最后几分钟所有这些变量值和度量值，查看自服务器启动后 688
的总值也同样有用。

接下来是对 SHOW STATUS 中所看到的各种变量的概述，但不是一个详尽的列表。对于给定变量的详情，最好查询 MySQL 用户手册，详见 *http://dev.mysql.com/doc/en/mysqld-option-tables.html*。当我们讨论一组以相同前缀开头的相关变量时，我们指的是“<前缀>_*”这样的变量。

注2： 最早在 *http://code.openark.org/blog/mysql/mysql-global-status-difference-using-single-query* 上发表。

线程和连接统计

这些变量用来跟踪尝试的连接、退出的连接、网络流量和线程统计。

- Connections, Max_used_connections, Threads_connected
- Aborted_clients, Aborted_connects
- Bytes_received, Bytes_sent
- Slow_launch_threads, Threads_cached, Threads_created, Threads_running

如果 Aborted_connects 不为 0，可能意味着网络有问题或某人尝试连接但失败（可能用户指定了错误的密码或无效的数据库，或某个监控系统正在打开 TCP 的 3306 端口来检测服务器是否活着）。如果这个值太高，可能有严重的副作用：导致 MySQL 阻塞一个主机。

Aborted_clients 有类似的名字但意思完全不同。如果这个值增长，一般意味着曾经有一个应用错误，例如程序在结束之前忘记正确地关闭 MySQL 连接。这一般并不表明有大问题。

二进制日志状态

Binlog_cache_use 和 Binlog_cache_disk_use 状态变量显示了在二进制日志缓存中有多少事务被存储过，以及多少事务因超过二进制日志缓存而必须存储到一个临时文件中。MySQL 5.5 还包含 Binlog_stmt_cache_use 和 Binlog_stmt_cache_disk_use，显示了非事务语句相应的度量值。所谓的"二进制日志缓存命中率"往往对配置二进制日志缓存的大小并没有参考意义。详细参考第 8 章中相关的话题。

689

命令计数器

Com_* 变量统计了每种类型的 SQL 或 C API 命令发起过的次数。例如，Com_select 统计了 SELECT 语句的数量，Com_change_db 统计一个连接的默认数据库被通过 USE 语句或 C API 调用更改的次数。Questions[注3] 变量统计总查询量和服务器收到的命令数。然而，它并不完全等于所有 Com_* 变量的总和，这与查询缓存命中、关闭和退出的连接，以及其他可能的因素有关。

Com_admin_commands 状态变量可能非常大。它不仅计数管理命令，并且还包括对 MySQL 实例的 Ping 请求。这些请求通过 C API 发起，并且一般来自客户端代码，例如下面的 Perl 代码。

注 3： 在 MySQL 5.1 中，这个变量被分解成 Questions 和 Queries，两者有轻微区别。

```
my $dbh = DBI->connect(...);
while ( $dbh && $dbh->ping ) {
   # Do something
}
```

这些 Ping 请求是“垃圾”查询。它们往往不会对服务器产生许多负载，但仍然是个浪费，因为网络回路时间会增加应用的响应时间。我们曾经看到 ORM 系统（Ruby on Rails 立即跃入脑海）在每次查询之前 Ping 服务器，而这是无意义的；Ping 服务器然后再查询是一个“跳跃之前看一下”设计模式的典型例子，它会产生竞争条件。我们同样看到过在每次查询之前更改默认库的数据库抽象函数库，这也会产生大量的 Com_change_db 命令。最好消除这两种做法。

临时文件和表

可以通过下列命令查看 MySQL 创建临时表和文件的计数。

```
mysql> SHOW GLOBAL STATUS LIKE 'Created_tmp%';
```

这显示了关于隐式临时表和文件的统计——执行查询时内部创建。在 Percona Server 中，同样有展示显式临时表（即由用户通过 CREATE TEMPORARY TABLE 所创建）的命令。

```
mysql> SHOW GLOBAL TEMPORARY TABLES;
```

句柄操作

690

句柄 API 是 MySQL 和存储引擎之间的接口。Handler_* 变量用于统计句柄操作，例如 MySQL 请求一个存储引擎来从一个索引中读取下一行的次数。可以通过下列命令查看这些变量。

```
mysql> SHOW GLOBAL STATUS LIKE 'Handler_%';
```

MyISAM 键缓冲

Key_* 变量包含度量值和关于 MyISAM 键缓冲的计数。可以通过下列命令查看这些变量。

```
mysql> SHOW GLOBAL STATUS LIKE 'Key_%';
```

文件描述符

如果你主要使用 MyISAM 存储引擎，那么 Open_* 变量揭示了 MySQL 每隔多久会打开每个表的 *.frm*、*.MYI* 和 *.MYD* 文件。InnoDB 保持所有的数据在表空间文件中，因此如果你主要使用 InnoDB，那么这些变量并不精确。可以通过下列命令查看 Open_* 变量。

```
mysql> SHOW GLOBAL STATUS LIKE 'Open_%';
```

查询缓存

通过查询 Qcache_* 状态变量可以检查查询缓存。

```
mysql> SHOW GLOBAL STATUS LIKE 'Qcache_%';
```

SELECT 类型

Select_* 变量是特定类型的 SELECT 查询的计数器。它们能帮助你了解使用各种查询计划的 SELECT 查询比率。不幸的是，并没有关于其他查询类型的状态变量，例如 UPDATE 和 REPLACE；然而，可以看一下 Handler_* 状态变量（前面讨论过）大致了解非 SELECT 查询的相对数量。要查看所有 Select_* 变量，使用下列命令。

```
mysql> SHOW GLOBAL STATUS LIKE 'Select_%';
```

以我们的判断，Select_* 状态变量可以按花费递增的顺序如下排列。

Select_range
: 在第一个表上扫描一个索引区间的联接数目。

Select_scan
: 扫描整个第一张表的联接数目。如果第一个表中每行都参与联接，这样计数并没有问题；如果你并不想要所有行但又没有索引以查找到所需要的行，那就糟糕了。

691 Select_full_range_join
: 使用在表 *n* 中的一个值来从表 *n*+1 中通过参考索引的区间内获取行所做的联接数。这个值或多或少比 Select_scan 开销多些，具体多少取决于查询。

Select_range_check
: 在表 *n*+1 中重新评估表 *n* 中的每一行的索引是否开销最小所做的联接数。这一般意味着在表 *n*+1 中对该联接而言并没有有用的索引。这个查询有非常高的额外开销。

Select_full_join
: 交叉联接或并没有条件匹配表中行的联接的数目。检测的行数是每个表中行数的乘积。这通常是个坏事情。

最后两个变量一般并不快速地增长，如果快速增长，则可能表明一个“糟糕”的查询引入到了系统中。具体可参考第 3 章中关于如何找到此类查询的讨论。

排序

在前面几章中我们已经讲了许多 MySQL 的排序优化，因此你应该知道排序是如何工作的。当 MySQL 不能使用一个索引来获取预先排序的行时，必须使用文件排序，这会增加 Sort_* 状态变量。除 Sort_merge_passes 外，你可以只是增加 MySQL 会用来排序的索引以改变这些值。Sort_merge_passes 依赖 sort_buffer_size 服务器变量（不要与 myisam_sort_buffer_size 服务器变量相混淆）。MySQL 使用排序缓冲来容纳排序的行块。当完成排序后，它将这些排序后的行合并到结果集中，增加 Sort_merge_passes，并且用下一个待排序的行块填充缓存。然而，使用这个变量来指导排序缓存的大小并不是个好方法，详情见第 3 章。

可以通过以下命令查看所有的 Sort_* 变量。

```
mysql> SHOW GLOBAL STATUS LIKE 'Sort_%';
```

当 MySQL 从文件排序结果中读取已经排好序的行并返回给客户端时，Sort_scan 和 Sort_range 变量会增长。不同点仅在于：前者是当查询计划导致 Select_scan 增加（参考前面的章节）时增加，而后者是当 Select_range 增加时增加。二者的实现和开销完全一样；仅仅指示了导致排序的查询计划类型。

表锁

692

Table_locks_immediate 和 Table_locks_waited 变量可告诉你有多少锁被立即授权，有多少锁需要等待。但请注意，它们只是展示了服务器级别锁的统计，并不是存储引擎级的锁统计。

InnoDB 相关

Innodb_*变量展示了 SHOW ENGINE INNODB STATUS 中包含的一些数据，本附录稍后会讨论。这些变量会按名字分组：Innodb_buffer_pool_*，Innodb_log_*，等等。稍后我们在检查完 SHOW ENGINE INNODB STATUS 后会更多地讨论 InnoDB 内幕。

这些变量存在于 MySQL 5.0 或更新版本中，它们有重要的副作用：它们会创建一个全局锁，然后在释放该锁之前遍历整个 InnoDB 缓冲池。同时，另外一些线程也会遇到该锁而阻塞，直到它被释放。这歪曲了一些状态值，比如 Threads_running，因此，它们看起来比平常更高（可能高许多，取决于系统此时有多忙）。当运行 SHOW ENGINE INNODB STATUS 或通过 INFORMATION_SCHEMA 表（在 MySQL 5.0 或更新版本中，SHOW STATUS 和 SHOW VARIABLES 与对 INFORMATION_SCHEMA 表的查询在幕后映射了起来）访问这些统计时，有相同的副作用。

因此，这些操作在这些版本的 MySQL 中会更加昂贵——检查服务器状态太频繁（例如，每秒一次）可能会显著增加负载。使用 `SHOW STATUS LIKE` 也无济于事，因为它要获取所有的状态然后再进行过滤。

MySQL 5.5 中相比 5.1 有更多的变量，在 Percona Server 中更多。

插件相关

MySQL 5.1 和更新的版本中支持可插拔的存储引擎，并在服务器内对存储引擎提供了注册它们自己的状态和配置变量的机制。如果你在使用一个可插拔的存储引擎，也许会看到许多插件特有的变量。类似的变量总是以插件名开头。

SHOW ENGINE INNODB STATUS

InnoDB 存储引擎在 `SHOW ENGINE INNODB STATUS` 输出中，老版本中对应的是 `SHOW INNODB STATUS`，显示出了大量的内部信息。

不像其他大部分 `SHOW` 命令，它的输出就是单独的一个字符串，没有行和列。它分为很多小段，每一段对应了 InnoDB 存储引擎不同部分的信息，其中有一些信息对于 InnoDB
693 开发者来说是非常有用的，但是，许多信息，如果你试着去理解，并且应用到高性能 InnoDB 调优的时候，你会发现它们非常有趣——甚至是非常必要的。

> 老版本的 InnoDB 经常把 64 位数字分成两部分来输出：高 32 位和低 32 位。有一个例子是事务 ID，比如 `TRANSACTION 0 3793469`，你可以这么来计算 64 位数字的值：把第一部分往左移动 32 位，然后加到第二部分上。我们在后面会展示几个例子。

输出内容包含了一些平均值的统计信息，例如 `fsync()` 每秒调用次数。这些平均值是自上次输出结果生成以来的统计数，因此，如果你正在检查这些值，那就要确保已经等待了 30s 左右的时间，使两次采样之间积累起足够长的统计时间并多次采样，检查计数器变化从而弄清其行为。并不是所有的输出都会在一个时间点上生成，因而也不是所有显示出来的平均值会在同一时间间隔里重新计算一遍。而且，InnoDB 有一个内部复位间隔，而它是不可预知的，各个版本也不一样。你应该检查一下输出，看看有哪些平均值在这个时间段里生成，因为每次采样的时间间隔不总是相同的。

这里面有足够的信息可供手工计算出大多数你想要的统计信息。但是，如果这时有一款监控工具，例如 *innotop*——它能为你计算出增量差值和平均值，那将是非常有用的。

头部信息

第一段是头部信息，它仅仅声明了输出开始，其内容包括当前的日期和时间，以及自上次输出以来经过的时长。下列第 2 行是当前日期和时间。第 4 行显示的是计算出这一平均值的时间间隔，即自上次输出以来的时间，或者是距离上次内部复位的时长。

```
1  =====================================
2  070913 10:31:48 INNODB MONITOR OUTPUT
3  =====================================
4  Per second averages calculated from the last 49 seconds
```

SEMAPHORES

如果有高并发的工作负载，你就要关注下接下来的段：SEMAPHORES（信号量）。它包含了两种数据：事件计数器，以及可选的当前等待线程的列表。如果有性能上的瓶颈，可以使用这些信息来找出瓶颈。不幸的是，想知道怎么使用这些信息还是有一点复杂，不过我们会在本附录的后面部分里给你一些建议。下面是一些输出样例。

694

```
1  ----------
2  SEMAPHORES
3  ----------
4  OS WAIT ARRAY INFO: reservation count 13569, signal count 11421
5  --Thread 1152170336 has waited at ./../include/buf0buf.ic line 630 for 0.00 second:
   the semaphore:
6  Mutex at 0x2a957858b8 created file buf0buf.c line 517, lock var 0
7  waiters flag 0
8  wait is ending
9  --Thread 1147709792 has waited at ./../include/buf0buf.ic line 630 for 0.00 second:
   the semaphore:
10 Mutex at 0x2a957858b8 created file buf0buf.c line 517, lock var 0
11 waiters flag 0
12 wait is ending
13 Mutex spin waits 5672442, rounds 3899888, OS waits 4719
14 RW-shared spins 5920, OS waits 2918; RW-excl spins 3463, OS waits 3163
```

第 4 行给出了关于操作系统等待数组的信息，它是一个“插槽”数组。InnoDB 在数组里为信号量保留了一些插槽，操作系统用这些信号量给线程发送信号，使线程可以继续运行，以完成它们等着做的事情。这一行还显示出 InnoDB 使用了多少次操作系统的等待。保留计数（reservation count）显示了 InnoDB 分配插槽的频度，而信号计数（signal count）衡量的是线程通过数组得到信号的频度。操作系统的等待相对于空转等待（spin wait）要更昂贵一些，我们即将看到这一点。

第 5 ~ 12 行显示的是当前正在等待互斥量的 InnoDB 线程。在这个例子里显示出有两个线程正在等待，每一个都是以“-- Thread <数字> has waited...”开始的。这一段应该是空的，除非服务器运行着高并发的工作负载，促使 InnoDB 采取让操作系统等待的措施。除非

你对 InnoDB 源代码很熟悉，否则这里看到的最有用的信息是发生线程等待的代码文件名。这就给了你一个提示：在 InnoDB 内部哪里才是热点。举例来说，如果看到许多线程都在一个名为 *buf0buf.ic* 的文件上等待着，那就意味着你的系统里存在着缓冲池竞争。这个输出信息还显示了这些线程等待了多长的时间，其中“waiters flag”显示了有多少个等待者正在等待同一个互斥量。

文本“wait is ending”意味着这个互斥量实际上已经被释放了，但操作系统还没把线程调度过来运行。

你可能想知道 InnoDB 真正等待的是什么。InnoDB 使用了互斥量和信号量来保护代码的临界区，例如，限定每次只能有一个线程进入临界区，或者是当有活动的读时，就限制写入等。在 InnoDB 代码里有很多临界区，在合适的条件下，它们都可能出现在那里。常常能见到的一种情形就是获取缓冲池分页的访问权。

在等待线程的列表之后，第 13 和 14 行显示了更多的事件计数器。第 13 行显示的是跟互斥量相关的几个计数器，第 14 行用于显示读 / 写共享和排他锁的计数器。在每一个情形中，都能看到 InnoDB 依靠操作系统等待的频度。

695 InnoDB 有着一个多阶段等待策略。首先，它会试着对锁进行空等待。如果经过了一个预设的空转等待周期（设置 `innodb_sync_spin_loops` 配置变量指令）之后还没有成功，那就会退到更昂贵更复杂的等待数组中。[注4]

空转等待的成本相对比较低，但是它们要不停地检查一个资源是否能被锁定，这种方式会消耗 CPU 周期。但是，这没有听起来那么糟糕，因为当处理器在等待 I/O 时，一般都有一些空闲的 CPU 周期可用，即使是没有空闲的 CPU 周期，空等也要比其他方式更加廉价一些。然而，当另外一条线程能做一些事情时，空转等待也还会独占处理器。

空转等待的替换方案就是让操作系统做上下文切换，这样，当这个线程在等待时，另外一个线程就可以被运行，然后，通过等待数组里的信号量发出信号，唤醒那个沉睡的线程。通过信号量来发送信号是比较有效率的，但是上下文切换就很昂贵，这很快就会积少成多：每秒钟几千次的切换会引发大量的系统开销。

你可以通过改变系统变量 `innodb_sync_spin_loops` 的值，试着在空转等待与操作系统等待之间达成平衡。不要担心空转等待，除非你在每一秒里会看到许多空转等待（大概是几十万这个水平）。这经常需要理解源代码或咨询专家才能解决。同样也可以考虑使用 Performance Schema，或看一下 `SHOW ENGINE INNODB MUTEX`。

注 4： 在 MySQL 5.1 中增强了等待数组，使其更为高效。

LATEST FOREIGN KEY ERROR

下一段，即 LATEST FOREIGN KEY ERROR，一般不会出现，除非你的服务器上有外键错误。在源代码里有许多地方会生成这样的输出，具体取决于错误的类型。有时问题在于事务在插入、更新或删除一条记录时要寻找到父行或子行。还有些时候是当 InnoDB 尝试增加或删除一个外键，或修改一个已经存在的外键时，发现表之间类型不匹配。

这部分输出对于调试与 InnoDB 往往不明确的外键错误相对应的准确原因非常有帮助。让我们看几个例子。首先，创建有外键关系的两个表，然后插入少量数据。

```
CREATE TABLE parent (
   parent_id int NOT NULL,
   PRIMARY KEY(parent_id)
) ENGINE=InnoDB;
CREATE TABLE child (
   parent_id int NOT NULL,
   KEY parent_id (parent_id),
   CONSTRAINT child_ibfk_1 FOREIGN KEY (parent_id) REFERENCES parent (parent_id)
) ENGINE=InnoDB;
INSERT INTO parent(parent_id) VALUES(1);
INSERT INTO child(parent_id) VALUES(1);
```
696

有两种基本的外键错误。以某种可能违反外键约束关系的方法增加、更新或删除数据，将导致第一类错误。例如，以下是当我们从父表中删除行时发生的事情。

```
DELETE FROM parent;
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint
fails (`test/child`, CONSTRAINT `child_ibfk_1` FOREIGN KEY (`parent_id`) REFERENCES
`parent` (`parent_id`))
```

错误信息相当直接明了，对所有由增加、更新或删除不匹配的行导致的错误都会看到相似的信息。下面是 SHOW ENGINE INNODB STATUS 的输出。

```
------------------------
LATEST FOREIGN KEY ERROR
------------------------
070913 10:57:34 Transaction:
TRANSACTION 0 3793469, ACTIVE 0 sec, process no 5488, OS thread id 1141152064
updating or deleting, thread declared inside InnoDB 499
mysql tables in use 1, locked 1
4 lock struct(s), heap size 1216, undo log entries 1
MySQL thread id 9, query id 305 localhost baron updating
DELETE FROM parent
Foreign key constraint fails for table `test/child`:
'
  CONSTRAINT `child_ibfk_1` FOREIGN KEY (`parent_id`) REFERENCES `parent` (`parent_
id`)
Trying to delete or update in parent table, in index `PRIMARY` tuple:
DATA TUPLE: 3 fields;
```

```
15  0: len 4; hex 80000001; asc     ;; 1: len 6; hex 00000039e23d; asc    9 =;; 2: len
   7; hex 000000002d0e24;  asc     - $;;
16
17 But in child table `test/child`, in index `parent_id`, there is a record:
18 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
19  0: len 4; hex 80000001; asc     ;; 1: len 6; hex 000000000500; asc     ;;
```

第 4 行显示了最近一次外键错误的日期和时间。第 5 ～ 9 行显示了关于破坏外键约束的事务详情。后面会再解释这些行。第 10 ～ 19 行显示了发现错误时 InnoDB 正尝试修改的准确数据。输出中有许多是转换成可打印格式的行数据。关于这点我们同样会在后面再加以说明。

到目前为止还没有什么问题，但有另外一类的外键错误，可能会让调试更难。以下是当我们尝试修改父表时所发生的。

```
ALTER TABLE parent MODIFY parent_id INT UNSIGNED NOT NULL;
ERROR 1025 (HY000): Error on rename of './test/#sql-1570_9' to './test/parent'
(errno: 150)
```

这就没有那么清楚了，但 SHOW ENGINE INNODB STATUS 的文本给了些指引信息。

697

```
------------------------
LATEST FOREIGN KEY ERROR
------------------------
070913 11:06:03 Error in foreign key constraint of table test/child:
there is no index in referenced table which would contain
the columns as the first columns, or the data types in the
referenced table do not match to the ones in table. Constraint:
,
  CONSTRAINT child_ibfk_1 FOREIGN KEY (parent_id) REFERENCES parent (parent_id)
The index in the foreign key in table is parent_id
See http://dev.mysql.com/doc/refman/5.0/en/innodb-foreign-key-constraints.html
for correct foreign key definition.
```

本例中的错误是数据类型不同。外键列必须有完全相同的数据类型，包括任何修饰符（例如本例中的 UNSIGNED，这也是问题所在）。当看到 1025 错误并不理解为什么时，最好查看 SHOW ENGINE INNODB STATUS。

在每次有新错误时，外键错误信息都会被重写。Percona Toolkit 中的 *pt-fk-error-logger* 工具可以保存这些信息以供后续分析。

LATEST DETECTED DEADLOCK

跟上面的外键部分一样，LATEST DETECTED DEADLOCK 部分也只有当服务器内有死锁时才会出现。死锁错误信息同样在每次有新错误时都会重写，Percona Toolkit 中的 *pt-deadlock-logger* 工具可以保存这些信息以供后续分析。

死锁在等待关系图里是一个循环，就是一个锁定了行的数据结构又在等待别的锁。这个循环可以任意地大。InnoDB 会立即检测到死锁，因为每当有事务等待行锁的时候，它都会去检查等待关系图里是否有循环。死锁的情况可能会比较复杂，但是，这一部分只显示了最近两个死锁的情况，它们在各自的事务里执行的最后一条语句，以及它们在图里形成循环锁的信息。在这个循环里你看不到其他事务，也看不到在事务里早先可能真正获得了锁的语句。尽管如此，通常还是可以通过查看这些输出结果来确定到底是什么引起了死锁。

在 InnoDB 里实际上有两种死锁。第一种就是人们常常碰到的那种，它在等待关系图里是一个真正的循环。另外一种就是在一个等待关系图里，因代价昂贵而无法检查它是不是包含了循环。如果 InnoDB 要在关系图里检查超过 100 万个锁，或者在检查过程中，InnoDB 要重做 200 个以上的事务，那它就会放弃，并宣布这里有一个死锁。这些数值都是硬编码在 InnoDB 代码里的常量，无法配置（如果你愿意，可以在代码里更改这些数值，然后重新编译）。当 InnoDB 的检查工作超过这个极限后，它就会引发一个死锁，这时你就可以在输出里看到一条信息“TOO DEEP OR LONG SEARCH IN THE LOCK TABLE WAITS-FOR GRAPH”。

InnoDB 不仅会打印出事务和事务持有及等待的锁，而且还有记录本身。这些信息主要 698
对于 InnoDB 开发者有用，但目前也没有办法禁止显示。不幸的是，它会变得很大，以致超过为输出结果预留的长度，使你无法看到下面几段输出信息。对此唯一的补救办法是，制造一个小的死锁来替换那个大的死锁，或者使用 Percona Server，该服务器软件增加了配置变量来抑制过于详尽的文本。

下面是一个死锁信息的样例。

```
------------------------
LATEST DETECTED DEADLOCK
------------------------
070913 11:14:21
*** (1) TRANSACTION:
TRANSACTION 0 3793488, ACTIVE 2 sec, process no 5488, OS thread id 1141287232
starting index read
mysql tables in use 1, locked 1
LOCK WAIT 4 lock struct(s), heap size 1216
MySQL thread id 11, query id 350 localhost baron Updating
UPDATE test.tiny_dl SET a = 0 WHERE a <> 0
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 3662 n bits 72 index `GEN_CLUST_INDEX` of table
`test/tiny_dl` trx id 0 3793488 lock_mode X waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 6; hex 000000000501 ...[ omitted ] ...

```

```
16 *** (2) TRANSACTION:
17 TRANSACTION 0 3793489, ACTIVE 2 sec, process no 5488, OS thread id 1141422400
   starting index read, thread declared inside InnoDB 500
18 mysql tables in use 1, locked 1
19 4 lock struct(s), heap size 1216
20 MySQL thread id 12, query id 351 localhost baron Updating
21 UPDATE test.tiny_dl SET a = 1 WHERE a <> 1
22 *** (2) HOLDS THE LOCK(S):
23 RECORD LOCKS space id 0 page no 3662 n bits 72 index `GEN_CLUST_INDEX` of table
   `test/tiny_dl` trx id 0 3793489 lock mode S
24 Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
25  0: ... [ omitted ] ...
26
27 *** (2) WAITING FOR THIS LOCK TO BE GRANTED:
28 RECORD LOCKS space id 0 page no 3662 n bits 72 index `GEN_CLUST_INDEX` of table
   `test/tiny_dl` trx id 0 3793489 lock_mode X waiting
29 Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
30  0: len 6; hex 000000000501 ...[ omitted ] ...
31
32 *** WE ROLL BACK TRANSACTION (2)
```

第 4 行显示的是死锁发生的时间，第 5 ~ 10 行显示的是死锁里的第一个事务的信息。在下一节里，我们会详尽地解释这些输出的含义。

699 第 11 ~ 15 行显示的是当死锁发生时，事务 1 正在等待的锁。我们忽略了其中第 14 行的信息，那是因为这只对调试才有用。这里要特别注意的内容是第 12 行，它告诉你这个事务正在等待对 test.tiny_dl 表中的 GEN_CLUST_INDEX[注5] 索引上排他锁（X 锁）。

第 16 ~ 21 行显示的是第二个事务的状态，第 22 ~ 26 行显示的是该事务持有的锁。为了简洁起见，在第 25 行有几条记录已经被我们删去了。这些记录里有一条就是第一个事务正在等待的那一条。最后，第 27 ~ 31 行显示了它正在等待的是哪一个锁。

当一个事务持有了其他事务需要的锁，同时又想获取其他事务持有的锁时，等待关系图上就会产生循环了。InnoDB 不会显示所有持有和等待的锁，但是，它显示了足够的信息来帮你确定：查询操作正在使用哪些索引。这对于你确定是否能避免死锁有着极大的价值。

如果能使两个查询对同一个索引朝同一个方向进行扫描，就能降低死锁的数目，因为，查询在同一顺序上请求锁的时候不会创建循环。有时候，这是很容易做到的，举例来说，如果要在一个事务里更新许多条记录，就可以在应用程序的内存里把它们按主键进行排序，然后，再用同样的顺序更新到数据库，这样就不会有死锁的发生。但是在另一些时候，这个方法也是行不通的（例如有两个进程使用了不同的索引区间操作同一张表的时候）。

第 32 行显示的是哪个事务被选中成为死锁的牺牲品。InnoDB 会把看上去最容易回滚(就

注 5： 这是在不指定主键时 InnoDB 内部创建的索引。

是更新的记录数最少的）的事务选为牺牲品。

检测这些常规日志，从中找出线程所涉及的查询，然后看一下到底是什么导致死锁，这非常有用。下节将介绍在哪里可以查找到死锁输出中的线程 ID。

TRANSACTIONS

本节包含了一些关于 InnoDB 事务的总结信息，紧随其后是当前活跃事务列表。以下是前几行信息（头部）。

```
1 ------------
2 TRANSACTIONS
3 ------------
4 Trx id counter 0 80157601
5 Purge done for trx's n:o <0 80154573 undo n:o <0 0
6 History list length 6
7 Total number of lock structs in row lock hash table 0
```

输出会因 MySQL 版本不同而变化，但至少包括如下几点。

- 第 4 行：当前事务的 ID，这是一个系统变量，每创建一个新事务都会增加。 700
- 第 5 行：这是 InnoDB 清除旧 MVCC 行时所用的事务 ID。将这个值和当前事务 ID 进行比较，可以知道有多少老版本的数据未被清除。这个数字多大才可以安全的取值没有硬性和速成的规定。如果数据没做过任何更新，那么一个巨大的数字也不意味着有未清除的数据，因为实际上所有事务在数据库里查看的都是同一个版本的数据。从另一方面来讲，如果有很多行被更新，那每一行就会有一个或多个版本留在内存里。减少此类开销的最好办法是确保事务一完成就立即将它提交，不要让它长时间地处于打开的状态。因为一个打开的事务即使不做任何操作，也会影响到 InnoDB 清理旧版本的行数据。
- 同样是在第 5 行里，还有一项 InnoDB 清理进程正在使用的撤销日志编号，如果有的话。如果它是“0 0”，如在本例中一样，说明清理进程处于空闲状态。
- 第 6 行：历史记录的长度，即位于 InnoDB 数据文件的撤销空间里的页面的数目。如果事务执行了更新并提交，这个数字就会增加；而当清理进程移除旧版本数据时，它就会递减。清理进程也会更新第 5 行中的数值。
- 第 7 行：锁结构的数目。每一个锁结构经常持有许多个行锁，所以，它跟被锁定行的数目不一样。

头部信息之后就是一个事务列表。当前版本的 MySQL 还不支持嵌套事务，因此，在某个时间点上，每个客户端连接能拥有的事务数目是有一个上限的，而且每一个事务只能属于单一连接。在输出信息里，每一个事务至少占有两行内容。下面这个例子就是关于一个事务所能看到的最少的信息。

```
1  ---TRANSACTION 0 3793494, not started, process no 5488, OS thread id 1141152064
2  MySQL thread id 15, query id 479 localhost baron
```

第 1 行以该事务的 ID 和状态开始。这个事务是“not started”，意思是已经提交并且没有再发起影响事务的语句；可能刚好空闲。然后是一些进程和线程信息。第 2 行显示了 MySQL 进程 ID，也和 SHOW FULL PROCESSLIST 中的 Id 列相同。紧随其后的是一个内部查询号和一些连接信息（同样与 SHOW FULL PROCESSLIST 中的相同）。

然而，每个事务会打印比这多得多的信息。下面是一个稍复杂一些的例子。

```
1  ---TRANSACTION 0 80157600, ACTIVE 4 sec, process no 3396, OS thread id 1148250464,
   thread declared inside InnoDB 442
2  mysql tables in use 1, locked 0
3  MySQL thread id 8079, query id 728899 localhost baron Sending data
4  select sql_calc_found_rows * from b limit 5
5  Trx read view will not see trx with id>= 0 80157601, sees <0 80157597
```

701 本例中的第 1 行显示此事务已经处于活跃状态 4s。可能的状态有“not started”、“active”、“prepared”和“committed in memory”（一旦被提交到磁盘上，状态就会变为“not started”）。尽管在这个示例里没有显示，但是在其他条件下，你也许能看到关于事务当前正在做什么的信息。在源代码中有超过 30 个字符串常量可以显示在这里，例如“fetching rows”、“adding foreign keys”，等等。

第 1 行里的文本“thread declared inside InnoDB 442”的意思是该线程正在 InnoDB 内核里做一些操作，并且，还有 442 张“票”可以使用。换句话说，就是同样的 SQL 查询可以重新进入 InnoDB 内核 442 次。这个“票”是系统用来限制内核中线程并发操作的手段，以防止其在某些平台上运行失常。即使线程的状态是“inside InnoDB”，它也不是在 InnoDB 里面完成所有的工作。查询可能是在服务器一级做一些操作，而只是通过某个途径跟 InnoDB 内核互动一下。你也可能看到事务的状态是“sleeping before joining InnoDB queue”或者“waiting in InnoDB queue”。

接下来一行显示了当前语句里有多少表被使用和锁定。InnoDB 一般不会锁定表，但对有些语句会锁定。如果 MySQL 服务器在高于 InnoDB 层次之上将表锁定，这里也是能够显示出来的。如果事务已经锁定了几行数据，这里将会有一行信息显示出锁定结构的数目（再声明一次，这跟行锁是两回事）和堆的大小。具体例子可以查看之前的死锁输出信息。在 MySQL 5.1 及更新的版本里，这一行还显示了当前事务持有的行锁的实际数目。

堆的大小指的是为了持有这些行锁而占用的内存大小。InnoDB 是用一种特殊的位图表来实现行锁的，从理论上讲，它可将每一个锁定的行表示为一个比特。我们的测试显示，每一个锁通常不超过 4 比特。

本例中的第 3 行包含的信息略微多于上例中的第 2 行：在该行的末尾是线程状态“Sending data”，这跟 SHOW FULL PROCESSLIST 中所看到的 Command 列相同。

如果事务正在运行一个查询，那么接下来就会显示出查询的文本（或者，在某些版本的 MySQL 里，显示其中的一小段），在本例中是第 4 行。

第 5 行显示了事务的读视图，它表明了因为版本关系而产生的对于事务可见和不可见两种类型的事务 ID 的范围。在本例中，在两个数字之间有一个四个事务的间隙，这四个事务可能是不可见的。InnoDB 在执行查询时，对于那些事务 ID 正好在这个间隙的行，还会检查其可见性。

如果事务正在等待一个锁，那么在查询内容之后将可以看到这个锁的信息。在上文的死锁例子里，这样的信息已经看到过多次了。不幸的是，输出信息并没有说出这个锁正被其他哪个事务持有。如果使用了 InnoDB 插件，就可以在 MySQL 5.1 及更高版本中的 INFORMATION-SCHEMA 表中查明这一点。

如果输出信息里有很多个事务，InnoDB 可能会限制要打印出来的事务数目，以免输出 702
信息增长得太大。这时就会看到“...truncated...”。

FILE I/O

FILE I/O 部分显示的是 I/O 辅助线程的状态，还有性能计数器的状态。

```
1  --------
2  FILE I/O
3  --------
4  I/O thread 0 state: waiting for i/o request (insert buffer thread)
5  I/O thread 1 state: waiting for i/o request (log thread)
6  I/O thread 2 state: waiting for i/o request (read thread)
7  I/O thread 3 state: waiting for i/o request (write thread)
8  Pending normal aio reads: 0, aio writes: 0,
9   ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0
10 Pending flushes (fsync) log: 0; buffer pool: 0
11 17909940 OS file reads, 22088963 OS file writes, 1743764 OS fsyncs
12 0.20 reads/s, 16384 avg bytes/read, 5.00 writes/s, 0.80 fsyncs/s
```

第 4 ~ 7 行显示了 I/O 辅助线程的状态。第 8 ~ 10 行显示的是每个辅助线程的挂起操作的数目，以及日志和缓冲池线程挂起的 fsync() 操作数目。简写“aio”的意思是“异步 I/O”。第 11 行显示了读、写和 fsync() 调用执行的数目。在你的负载下这些绝对值会有所不同，因此更重要的是监控它们过去一段时间内是如何改变的。第 12 行显示了在头部显示的时间段内的每秒平均值。

在第 8 ~ 9 行显示的挂起值是检测 I/O 受限的应用的一个好方法。如果这些 I/O 大部分

有挂起的操作，那么负载可能 I/O 受限。

在 Windows 下，可以通过 `innodb_file_io_threads` 配置变量来调整 I/O 辅助线程数，因此可能会看到不止一个读线程和写线程。在使用了 InnoDB 插件的 MySQL 5.1 和更新版本中，或 Percona Server 中，可以使用 `innodb_read_io_threads` 和 `innodb_write_io_threads` 来为读 / 写配置多个线程。然而，在所有平台下至少总会看到 4 个线程。

Insert buffer thread

负责插入缓冲合并（例如，记录被从插入缓冲合并到表空间中）。

Log thread

负责异步刷日志。

703 *Read thread*

执行预读操作以尝试预先读取 InnoDB 预感需要的数据。

Write thread

刷脏缓冲。

INSERT BUFFER AND ADAPTIVE HASH INDEX

这部分显示了 InnoDB 内这两个结构的状态。

```
1  -------------------------------------
2  INSERT BUFFER AND ADAPTIVE HASH INDEX
3  -------------------------------------
4  Ibuf for space 0: size 1, free list len 887, seg size 889, is not empty
5  Ibuf for space 0: size 1, free list len 887, seg size 889,
6  2431891 inserts, 2672643 merged recs, 1059730 merges
7  Hash table size 8850487, used cells 2381348, node heap has 4091 buffer(s)
8  2208.17 hash searches/s, 175.05 non-hash searches/s
```

第 4 行显示了关于插入缓存大小、“free list” 的长度和段大小的信息。文本 “for space 0” 像是指明了多个插入缓冲的可能性——每个表空间一个，但从未实现，并且这个文本在最近的 MySQL 版本中被移除掉了。只有一个插入缓冲，因此第 5 行真的是多余的。第 6 行显示了有多少缓冲操作已经完成。合并与插入的比例很好地说明了缓冲使用效率如何。

第 7 行显示了自适应哈希索引的状态。第 8 行显示了在头部提及的时间内 InnoDB 完成了多少哈希索引操作。哈希索引查找与非哈希索引查找的比例仅供参考。自适应索引无法配置。

LOG

这部分显示了关于 InnoDB 事务日志（重做日志）子系统的统计。

```
1 ---
2 LOG
3 ---
4 Log sequence number 84 3000620880
5 Log flushed up to   84 3000611265
6 Last checkpoint at  84 2939889199
7 0 pending log writes, 0 pending chkp writes
8 14073669 log i/o's done, 10.90 log i/o's/second
```

第 4 行显示了当前日志序号，第 5 行显示了日志已经刷到哪个位置。日志序号就是写到日志文件中的字节数，因此可用来计算日志缓冲中还有多少没有写入到日志文件中。在这个例子中，它有 9615 字节（13 000 620 880 − 13 000 611 265）。第 6 行显示了上一检测点（一个检测点表示一个数据和日志文件都处于已知状态的时刻，并且能用于恢复）。 704 如果上一检查点落后日志序号太多，并且差异接近于该日志文件的大小，InnoDB 会触发“疯狂刷”，这对性能而言非常糟糕。第 7 ~ 8 行显示了挂起的日志操作和统计，你可以将其与 `FILE I/O` 部分的值相比较，以了解你的 I/O 有多少是由日志子系统引起，有多少是其他原因。

BUFFER POOL AND MEMORY

这部分显示了关于 InnoDB 缓冲池及其如何使用内存的统计。

```
1  ----------------------
2  BUFFER POOL AND MEMORY
3  ----------------------
4  Total memory allocated 4648979546; in additional pool allocated 16773888
5  Buffer pool size   262144
6  Free buffers       0
7  Database pages      258053
8  Modified db pages  37491
9  Pending reads 0
10 Pending writes: LRU 0, flush list 0, single page 0
11 Pages read 57973114, created 251137, written 10761167
12 9.79 reads/s, 0.31 creates/s, 6.00 writes/s
13 Buffer pool hit rate 999 / 1000
```

第 4 行显示了由 InnoDB 分配的总内存，以及其中多少是额外内存池分配。额外内存池仅分配了其中（一般很小）一部分的内存，由内部内存分配器分配。现代的 InnoDB 版本一般使用操作系统的内存分配器，但老版本使用自己的，这是由于在那个时代有些操作系统并未提供一个非常好的实现。

第 5 ~ 8 行显示了缓冲池度量值，以页为单位。度量值有总的缓冲池大小、空闲页数、

分配用来存储数据库页的页数，以及“脏”数据库页数。InnoDB 使用缓冲池中的部分页来对锁、自适应哈希，以及其他系统结构做索引，因此池中的数据库页数永远不等于总的池大小。

第 9 ~ 10 行显示了挂起的读和写的数量（例如 InnoDB 需要为缓冲池而做的总的逻辑读和写）。这些值并不与 `FILE I/O` 部分的值相匹配，因为 InnoDB 可能合并许多的逻辑操作到一个物理 I/O 操作中。LRU 代表“最近使用到的”；它是通过冲刷缓冲中不经常使用的页来释放空间以供给经常使用的页的一种方法。冲刷列表存放有检测点处理需要冲刷的旧页，并且单页的写是独立的页面写，不会被合并。

输出中的第 8 行显示缓冲池包含 37 491 个脏页，这是在某些时刻（它们已经在内存中被修改但尚未写到磁盘上）需要被刷到磁盘上的。然而，第 10 行显示当前没有安排冲刷。
705 这不是一个问题；InnoDB 会在需要时刷。如果在 InnoDB 的状态输出中到处可见大量挂起的 I/O 操作，这往往表明服务器有严重问题。

第 11 行显示了 InnoDB 被读取、创建和写入了多少页。读 / 写页的值指的是从磁盘读到缓冲池中的数据，或反过来说。创建页的值指的是 InnoDB 在缓冲池中分配但没有从数据文件中读取内容的页，因为它并不关心内容是什么（例如，它们可能属于一个已经被删除的表）。

第 13 行报告了缓冲池的命中率，它用来衡量 InnoDB 在缓冲池中查找到所需页的比例。它度量自上次 InnoDB 状态输出后的命中率，因此，如果服务器自那以后一直很安静，你将会看到“No buffer pool page gets since the last printout.”。它对于度量缓存池的大小并没有用处。

在 MySQL 5.5 中，可能有多个缓冲池，每一个都会在输出中打印一部分信息。Percona XtraDB 还会在输出中打印更多详情——例如，准确显示内存在哪里分配。

ROW OPERATIONS

这部分显示了其他各项 InnoDB 统计。

```
--------------
ROW OPERATIONS
--------------
0 queries inside InnoDB, 0 queries in queue
1 read views open inside InnoDB
Main thread process no. 10099, id 88021936, state: waiting for server activity
Number of rows inserted 143, updated 3000041, deleted 0, read 24865563
0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
----------------------------
END OF INNODB MONITOR OUTPUT
============================
```

第 4 行显示了 InnoDB 内核内有多少线程（我们在讨论 TRANSACTIONS 部分的小节中提及过）。队列中的查询是 InnoDB 为限制并发执行的线程量而不允许进入内核的线程。查询同样在进入队列之前会休眠等待，这之前已经讨论过。

第 5 行显示了有多少打开的 InnoDB 读视图。读视图是包含事务开始点的数据库内容的 MVCC “快照”。你可以看看某特定事务是否在 TRANSACTIONS 部分有读视图。

第 6 行显示了内核的主线程状态。可能的状态值如下。

- doing background drop tables
- doing insert buffer merge
- flushing buffer pool pages
- flushing log 706
- making checkpoint
- purging
- reserving kernel mutex
- sleeping
- suspending
- waiting for buffer pool flush to end
- waiting for server activity

在大部分服务器上应该会经常看到 “sleeping”，如果生成多个快照而一再查看到不同的状态，例如“flushing buffer pool pages”，则应该怀疑相关的活动有问题——例如，“疯狂刷”问题，可能由某个冲刷算法差劲的 InnoDB 版本引起，或由糟糕的配置导致，例如太小的事务日志文件。

第 7 ~ 8 行显示了多少行被插入、更新、删除和读取，以及它们的每秒均值。如果想查看 InnoDB 有多少工作在进行，那么它们是很好的参考值。

SHOW ENGINE INNODB STATUS 输出在第 9 ~ 13 行结束。如果看不到这个文本，那可能是有一个大的死锁截断了输出。

SHOW PROCESSLIST

进程列表是当前连接到 MySQL 的连接或线程的清单。SHOW PROCESSLIST 列出了这些线程，以及每个线程的状态信息。例如：

```
mysql> SHOW FULL PROCESSLIST\G
*************************** 1. row ***************************
     Id: 61539
   User: sphinx
   Host: se02:58392
     db: art136
Command: Query
   Time: 0
  State: Sending data
   Info: SELECT a.id id, a.site_id site_id, unix_timestamp(inserted) AS
inserted,forum_id, unix_timestamp(p
*************************** 2. row ***************************
     Id: 65094
   User: mailboxer
   Host: db01:59659
     db: link84
Command: Killed
   Time: 12931
  State: end
707   Info: update link84.link_in84 set url_to =
replace(replace(url_to,'&amp;','&'),'%20','+'), url_prefix=repl
```

有几个工具（例如 *innotop*）可以以定期刷新的方式显示进程列表。

也可以从 INFORMATION_SCHEMA 中的表来获取这个信息。Percona Server 和 MariaDB 向这个表中增加了更多有用的信息，如高精度的时间字段和显示查询完成百分比的字段，这一信息可用作进度指示。

Command 和 State 列真正表明了线程的状态。上面的例子中，第一个进程正在运行查询并发送数据，而第二个进程已被杀死，这可能是由于这需要非常长的一段时间来完成，于是某人深思熟虑后通过 KILL 命令终结了它。线程有可能在 KILL 状态停留一段时间，因为 KILL 命令有可能不能立刻执行完成，比如它可能需要一些时间来回滚事务。

SHOW FULL PROCESSLIST（增加了 FULL 关键字）将显示每个查询的全文，否则最多显示 100 个字符。

SHOW ENGINE INNODB MUTEX

SHOW ENGINE INNODB MUTEX 返回 InnoDB 互斥体的详细信息，主要对洞悉可扩展性和并发性问题有帮助。每个互斥体都保护着代码中一个临界区，这在之前已经讨论过。

输出会因 MySQL 版本和编译选项而有所不同。下面是 MySQL 5.5 服务器的示例。

```
mysql> SHOW ENGINE INNODB MUTEX;
+--------+-----------------------------+------------+
| Type   | Name                        | Status     |
+--------+-----------------------------+------------+
| InnoDB | &table->autoinc_mutex       | os_waits=1  |
| InnoDB | &table->autoinc_mutex       | os_waits=1  |
| InnoDB | &table->autoinc_mutex       | os_waits=4  |
| InnoDB | &table->autoinc_mutex       | os_waits=1  |
| InnoDB | &table->autoinc_mutex       | os_waits=12 |
| InnoDB | &dict_sys->mutex            | os_waits=1  |
| InnoDB | &log_sys->mutex             | os_waits=12 |
| InnoDB | &fil_system->mutex          | os_waits=11 |
| InnoDB | &kernel_mutex               | os_waits=1  |
| InnoDB | &dict_table_stats_latches[i] | os_waits=2  |
| InnoDB | &dict_table_stats_latches[i] | os_waits=54 |
| InnoDB | &dict_table_stats_latches[i] | os_waits=1  |
| InnoDB | &dict_table_stats_latches[i] | os_waits=31 |
| InnoDB | &dict_table_stats_latches[i] | os_waits=41 |
| InnoDB | &dict_table_stats_latches[i] | os_waits=12 |
| InnoDB | &dict_table_stats_latches[i] | os_waits=1  |
| InnoDB | &dict_table_stats_latches[i] | os_waits=90 |
| InnoDB | &dict_table_stats_latches[i] | os_waits=1  |
| InnoDB | &dict_operation_lock        | os_waits=13 |
| InnoDB | &log_sys->checkpoint_lock   | os_waits=66 |
| InnoDB | combined &block->lock       | os_waits=2  |
+--------+-----------------------------+------------+
```

708

基于等待的数量，可以使用这个输出来帮助确定 InnoDB 的哪一块是瓶颈。只要有互斥体，就会有潜在的争用。该命令的输出可能会非常多，需要写一些脚本进行聚合分析。

有三种主要的策略可以消除互斥相关的瓶颈：尽量避开 InnoDB 的弱点，限制并发，或者在 CPU 密集型的空转等待和资源密集型的操作系统等待之间取得平衡。这些在本附录前面和第 8 章讨论过。

复制状态

MySQL 有几个命令用以监测复制。在主库上执行 SHOW MASTER STATUS 可显示主库的复制状态和配置。

```
mysql> SHOW MASTER STATUS\G
*************************** 1. row ***************************
            File: mysql-bin.000079
        Position: 13847
    Binlog_Do_DB:
Binlog_Ignore_DB:
```

输出包含了主库当前的二进制日志位置。通过 SHOW BINARY LOGS 可以获取到二进制日志的列表。

```
mysql> SHOW BINARY LOGS
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| mysql-bin.000044 |     13677 |
...
| mysql-bin.000079 |     13847 |
+------------------+-----------+
36 rows in set (0.18 sec)
```

要查看这些二进制日志中的事件，可以用 SHOW BINLOG EVENTS。在 MySQL 5.5 中，也可以使用 SHOW RELAYLOG EVENTS。

在备库上执行 SHOW SLAVE STATUS 查看复制的状态和配置。在此，我们不予列举，因为输出有点冗长，但我们会说明关于它的几个事情。首先，你可以同时看到复制 I/O 和复制 SQL 线程的状态，包括任何错误。也可以看到复制落后多远。输出中还有三套二级制日志的坐标，这几个坐标对于备份和搭备库非常有用。

Master_Log_File/Read_Master_Log_Pos
: I/O 线程读主库二进制日志的位置。

709 Relay_Log_File/Relay_Log_Pos
: SQL 线程执行中继日志的位置。

Relay_Master_Log_File/Exec_Master_Log_Pos
: SQL 线程执行的映射到主库二进制日志的位置。这与 Relay_Log_File/Relay_Log_Pos 有着相同的逻辑位置，但是主库的二进制日志而非复制的中继日志。换句话说，如果你看一下日志中的这两个位置，你会发现有相同的日志事件。

INFORMATION_SCHEMA

INFORMATION_SCHEMA 库是一个 SQL 标准中定义的系统视图的集合。MySQL 实现了许多标准中的视图，并且增加了一些其他的视图。在 MySQL 5.1 中，其中许多的视图与 MySQL 的 SHOW 命令对应，例如 SHOW FULL PROCESSLIST 和 SHOW STATUS。然而，也有一些视图并没有相对应的 SHOW 命令。

INFORMATION_SCHEMA 视图的美在于能够以标准的 SQL 来进行查询。这比 SHOW 命令更灵活，因为 SHOW 命令产生的结果不能聚合、联接或进行其他标准 SQL 操作。在系统视图层拥有所有可获得的数据使得写感兴趣和有用的查询变得可行。

例如，在 Sakila 样本库中哪一个表引用了 actor 表？一致的命名约定使之很容易确定。

```
mysql> SELECT TABLE_NAME FROM INFORMATION_SCHEMA.COLUMNS
    -> WHERE TABLE_SCHEMA='sakila' AND COLUMN_NAME='actor_id'
    -> AND TABLE_NAME <> 'actor';
+------------+
| TABLE_NAME |
+------------+
| actor_info |
| film_actor |
+------------+
```

我们需要为本书找几个表中含有多列索引的样例。下面是一个满足需要的查询。

```
mysql>  SELECT TABLE_NAME, GROUP_CONCAT(COLUMN_NAME)
    ->  FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
    ->  WHERE TABLE_SCHEMA='sakila'
    ->  GROUP BY TABLE_NAME, CONSTRAINT_NAME
    ->  HAVING COUNT(*) > 1;
+---------------+------------------------------------+
| TABLE_NAME    | GROUP_CONCAT(COLUMN_NAME)          |
+---------------+------------------------------------+
| film_actor    | actor_id,film_id                   |
| film_category | film_id,category_id                |
| rental        | customer_id,rental_date,inventory_id |
+---------------+------------------------------------+
```

你也可以写更复杂的查询，就像对待其他常规表一样。MySQL Forge（*http://forge.mysql.com*）是一个寻找和分享针对这些视图的查询的好地方。有查找重复和冗余索引，查找非常低基数的索引，以及更多其他的例子。在 Shlomi Noach 的 *common_schema* 项目中（*http://code.openark.org/forge/common_schema*）中同样有一组基于 INFORMATION_SCHEMA 视图所写的有用视图。 710

最大的缺点是视图与相应的 SHOW 命令相比，有时非常慢。它们一般会取所有的数据，存在临时表中，然后使查询可以获取临时表。当服务器上数据量大或表非常多时，查询 INFORMATION_SCHEMA 表会导致非常高的负载，并且会导致服务器对其他用户而言停转或不可响应，因此在一个高负载且数据量大的生产服务器上使用时要小心。查询时会有危险的表主要是那些包含下列表元数据的表：TABLES，COLUMNS，REFERENTIAL_CONSTRAINTS，KEY_COLUMN_USAGE，等等。对这些表的查询会导致 MySQL 向存储引擎请求获取类似服务器上表的索引统计等数据，而这在 InnoDB 里是非常繁重的。

这些视图不可更新。尽管你可以从中检索到服务器设置，但不能更新以影响服务器的配置，因此，仍然需要对配置使用 SHOW 和 SET 命令，尽管 INFORMATION_SCHEMA 视图对其他任务非常有用。

InnoDB 表

在 MySQL 5.1 和更新版本中，InnoDB 插件创建了许多的 INFORMATION_SCHEMA 表。这些表非常有用。在 MySQL 5.5 中有更多这样的表，而还未发行的 MySQL 5.6 中则还要多。

在 MySQL 5.1 中，存在如下一些表。

INNODB_CMP 和 INNODB_CMP_RESET

  这些表显示了 InnoDB 中以新文件格式 Barracuda 压缩的数据的相关信息。第二个表显示的信息与第一个表相同，但具有重置所包含数据的副作用，好像使用 FLUSH 命令那样。

INNODB_CMPMEM 和 INNODB_CMPMEM_RESET

  这些表显示了用于InnoDB压缩数据的缓冲池中页的信息。第二个表又是一个重置表。

INNODB_TRX 和 INNODB_LOCKS

  这些表显示了事务，拥有和等待锁的事务。它们对于诊断锁等待问题和长时间运行的事务非常重要。MySQL 用户手册上包含了查询样例，你可以直接复制、粘贴来显示哪一些事务在阻塞其他事务，它们正在运行的查询，等等。

711 除了这些表，MySQl 5.5 还增加了 INNODB_LOCK_WAITS，它可以帮助更容易地诊断更多类型的锁等待问题。MySQL 5.6 中将会增加显示关于 InnoDB 内部更多信息（包括缓冲池和数据字典）的表，以及称为 INNODB_METRICS 的新表，它将是使用 Performance Schema 的替代方案。

Percona Server 中的表

Percona Server 向 INFORMATION_SCHEMA 库中增加了大量的表。原生的 MySQL 5.5 服务器有 39 个表，而 Percona Server 5.5 有 61 个表。以下是关于新增表的概述。

“用户统计信息”表

  这些表源于 Google 的 MySQL 补丁。它们显示了客户端、索引、表、线程和用户的活动统计。我们在本书中提到了它们的使用，例如确定复制何时开始接近追赶上主库的能力极限。

InnoDB 数据字典

  一系列的表以只读表的方式暴露了 InnoDB 内部数据词典：列、外键、索引、统计，等等。它们对从 InnoDB 角度检测和理解数据库非常有帮助，它可能与 MySQL 不同，因为 MySQL 依赖于 *.frm* 文件来存储数据字典。类似的表在 MySQL 5.6 发行时会加进来。

InnoDB 缓冲池

这些表使你可以像表一样查询缓冲池，表中每个页是一行，因此，你可以看到什么页驻存于缓冲池中，有哪种类型的页，等等。这些表已被证实对于诊断类似膨胀的插入缓冲非常有用。

临时表

这些表显示了与 INFORMATION_SCHEMA.TABLES 表中可获取的类型相同的信息，只是用临时表取代了。有一个用于你自身会话的临时表，还有一个用于整个服务器中的所有临时表。它们对某个会话获取可视性到存在的临时表中，以及它们使用了多少空间。

杂项表

有少数其他表为查询执行时间、文件、表空间和更多 InnoDB 内部信息增加了可视性。

关于 Percona Server 的新增表的文档可以在 *http://www.percona.com/doc/* 获取。

Performance Schema 712

自 MySQL 5.5 起，Performance Schema（寄存于 PERFORMANCE_SCHEMA 库中）是 MySQL 增强仪表的新的汇总处。我们在第 3 章中已讨论过一点。

默认情况下，Performance Schema 是禁掉的，你必须打开并且使其在一个想要收集的特定的仪表点（“消费者”）启用。我们对服务器以几个不同的配置做了基准测试，发现即使 Performance Schema 没有数据可采集也会导致 8% ~ 11% 的开销，并且所有消费都生效的话会有 19% ~ 25% 的开销，具体取决于是一个只读还是读 / 写的负载。这算少还是多由你来决定。

这在 MySQL 5.6 中将改善，特别是当特性本身生效但所有仪表点都禁用时。这对某些用户而言更加实用，他们会让 Performance Schema 生效，但直到收集信息时才将其激活。

在 MySQL 5.5 中，Performance Schema 包含了指示条件变量、互斥体、读 / 写锁和文件 I/O 实例的表。还有指示实例上的等待信息的表，而这些经常是你在查询时首先感兴趣的，以及与其实例表的联接。这些事件等待表有几种变体，拥有关于服务器性能和行为的当前和历史信息。最后，还有一组“设置表”，你可以用这些表来使预想的消费者生效或失效。

在 MySQL 5.6.3 开发里程碑的第 6 个发行中，Performance Schema 中的表数从 17 增长到了 49。这意味着 MySQL 5.6 中有许多的仪表！增加的仪表涵盖 SQL 语句、语句过程（基本上与你在 SHOW PROCESSLIST 中看到的线程状态相同）、表、索引、主机、线程、用户、账号，以及各种总述及历史表等。

你如何使用这些表？有 49 个表，得让某些人为此写些工具来帮助大家了。然而，对于与 Performance Schema 表相对应的早期流行的非常不错的 SQL 例子，可以阅读 Oracle 工程师 Mark Leith 的博客上的一些文章，例如 *http://www.markleith.co.uk/?p=471*。

713 总结

MySQL 暴露服务器内部信息的首要方式是 SHOW 命令，但这在改变。在 MySQL 5.1 中引入的可插拔的 INFORMATION_SCHEMA 表允许 InnoDB 插件增加一些非常有意义的仪表，而 Percona Server 增加的要多得多。然而，读取 SHOW ENGINE INNODB STATUS 输出并解释的能力对管理 InnoDB 仍然是至关重要的。在 MySQL 5.5 和更新的服务器版本中，可以使用 Performance Schema，它将来可能变成深入服务器内部最强大和完备的方式。Performance Schema 最棒的一点在于它是基于时间的，这意味着 MySQL 最终可以获取已经逝去时间里的仪表盘，而不仅仅是已操作的次数。

附录 C

大文件传输

在管理 MySQL、初始化服务器、克隆复制和进行备份 / 还原操作时，复制、压缩和解压缩大文件（常常是跨网络的）是很常见的任务。能够最快最好完成这些任务的方法并不总是显而易见的，并且方法好坏的差异可能非常显著。这个附录将通过几个例子演示使用常见的 UNIX 实用工具，将一个大尺寸的备份镜像从一台服务器复制到其他服务器。

通常从未压缩的文件开始，例如一台服务器上的 InnoDB 表空间和日志文件。当然，在把文件复制到目的地之后要再将它解压缩。另外一个常见的场景是以压缩文件开始，例如备份镜像文件，以解压文件结束。

如果网络传输能力有限，那么用压缩格式在网络间发送文件是个好方法。你可能还需要一个安全的传输途径，使数据不会被损坏；这对于备份镜像文件来说，是一个很常见的需求。

复制文件

这个任务实际上就是完成以下事情。

1. （可选）压缩数据。
2. 发送到另外一台机器上。
3. 把数据解压缩到最终目的地。
4. 在复制完成后，校验文件以确认其没有被损坏。

我们对能达成这些目标的一系列方法进行了基准测试。本附录的余下部分将展示我们是怎么做的，以及我们找到的最快速的方法是什么。

对于在本书里讨论过的很多目的，例如备份，你可能要考虑在哪一台机器上做压缩会更

好一点。如果有足够的网络带宽，还是复制未压缩形式的备份镜像文件为好，这样可以在 MySQL 服务器上节省出 CPU 资源供查询使用。

716

一个简单的示例

我们以一个简单的示例开始，安全地将一个未压缩的文件从一台机器发送到另外一台上，途中将它进行压缩，然后再解压。在称为 server1 的源服务器上，执行如下命令。

```
server1$ gzip -c /backup/mydb/mytable.MYD > mytable.MYD.gz
server1$ scp mytable.MYD.gz root@server2:/var/lib/myql/mydb/
```

然后，在 server2 上执行如下命令。

```
server2$ gunzip /var/lib/mysql/mydb/mytable.MYD.gz
```

这大概是最简单的实现方法了，但效率并不高，因为涉及压缩、复制和解压缩等串行化的步骤。每一个步骤都需要读 / 写磁盘，速度比较慢。上述命令的真正操作依次是这样的：在 server1 上 *gzip* 既要读又要写，*scp* 在 server1 上读而在 server2 上写；*gunzip* 在 server2 上既要读又要写。

一步到位的方法

下面这个方法更有效率一些，它将压缩、复制文件和在传输的另一端解压缩文件全部放在一个步骤里完成。这一次我们使用 SSH，SCP 就是基于这个安全协议的。下面是在 server1 上执行的命令。

```
server1$ gzip -c /backup/mydb/mytable.MYD | ssh root@server2 "gunzip -c - > /var/lib
>/mysql/mydb/mytable.MYD"
```

这个方法通常比第一个方法好，因为它极大地降低了磁盘 I/O：磁盘活动被减少到只要在 server1 上读，在 server2 上写。这也使得磁盘操作更加有序。

也可以使用 SSH 内建的压缩来完成，但是我们展示的是用管道来做压缩和解压缩，这是因为这样能给予你更大的灵活性。例如，假如你不想在另一端解压缩文件，就无法使用 SSH 的压缩。

可以通过调整一些选项来提高这个方法的效率，例如给 *gzip* 增加选项 *-1*，使其压缩得更快。这个选项通常不会降低太多压缩率，但是能明显提高压缩速度，这才是重点。你也可以使用不同的压缩算法。例如，如果想获得很高的压缩率，又不在乎会花费多少时间，那么，就可以使用 *bzip2* 来代替 *gzip*。如果想要非常快的压缩速度，可以使用基于 LZO 的压缩程序。这样压缩后的数据会比其他方法的结果大 20% 左右，但是压缩的速度约快 5 倍。

避免加密的系统开销

SSH 不是跨网传输数据的最快方法，因为它增加了加解密的系统开销。如果不需要加密，那就使用 *netcat* 把“裸”数据进行跨网复制。可以通过 *nc* 以非交互式操作方式调用这个工具，这正是我们想要的。 717

这里有一个例子。首先，在 server2 上监听 12345 端口（任何闲置的端口都可以）上的文件，把任何发送到该端口的东西都解压缩到期望的数据文件里。

```
server2$ nc -l -p 12345 | gunzip -c - > /var/lib/mysql/mydb/mytable.MYD
```

然后在 server1 上，开启另一个 *netcat* 实例，发送数据到目的服务器监听的端口上。*-q* 选项告诉 *netcat* 当到达输入文件的末尾后就关闭连接。这会触发监听实例关闭接收的文件并退出。

```
server1$ gzip -c - /var/lib/mysql/mydb/mytable.MYD | nc -q 1 server2 12345
```

更容易的技术是使用 *tar*，这样文件名称也会通过网络发送出去，从而消除了另一个错误的来源，并会自动将文件写到正确的位置。*z* 选项告诉 *tar* 使用 *gzip* 做压缩和解压缩。下面是在 server2 上执行的命令。

```
server2$ nc -l -p 12345 | tar xvzf -
```

以下是在 server1 上执行的命令。

```
server1$ tar cvzf - /var/lib/mysql/mydb/mytable.MYD | nc -q 1 server2 12345
```

你可以把这些命令集成到一个单独的脚本里，这样压缩和复制大量的文件到网络连接时效率会比较高，然后在另一端解压缩。

其他选项

另外一个选择是 *rsync*。*rsync* 非常简便，因为它易于在源和目标之间做镜像，并且还可以断点续传。但是，当它的二进制差异算法无法被很好地发挥时，它不太会得到很好应用。在知道文件中的大部分内容都不需要传输的场景下，例如，如果要续传一个中途退出的 *nc* 复制的任务，就可以考虑用它。

在还没有处于危急关头时就应该针对文件传输做一些实验，因为发现哪一种方法最快可能要做许多试验和遇到许多错误。哪一种方法最快取决于你的系统。其中最大的影响因素是服务器上的磁盘驱动器、网卡和 CPU 的数量，以及它们之间相对的速度有多快。有个不错的方法是监控 *vmstat -n 5*，看磁盘或 CPU 是否就是速度的瓶颈。

如果有闲置的 CPU，就可能通过运行并行复制操作来加快整个过程。相反，如果 CPU 已经是瓶颈，而磁盘和网络的承载能力还比较充裕，那就可以不压缩。在导出和还原时，出于速度的考虑，并行执行这些操作往往是个不错的主意。此外，监控服务器性能，看看是否还有闲置的承载能力。过度的并行反而会降低处理速度。

718 文件复制的基准测试

为了便于比较，表 C-1 显示的是在局域网里通过一块标准的百兆以太网链路复制一个样本文件能达到的最快速度。这个文件未压缩时的大小是 738MB，使用 *gzip* 默认选项压缩后是 100MB。源和目的机器都有充足的可用内存、CPU 资源和磁盘空间；网络是瓶颈所在。

表C-1：跨网复制文件的基准测试

方法	时间（s）
rsync，不使用压缩	71
scp，不使用压缩	68
nc，不使用压缩	67
rsync，使用压缩（*-z*）	63
gzip，*scp* 和 *gunzip*	60(44+10+6)
ssh，使用压缩	44
nc，使用压缩	42

注意通过网络发送文件时压缩有多大的帮助——最慢的三个方法并没有压缩文件。尽管这样，好处也不一。如果 CPU 和磁盘慢但有一个千兆以太网连接，那么读取和压缩文件可能是瓶颈，不压缩反而更快。

顺便提一下，使用类似 *gzip --fast* 的快速压缩比默认压缩级别要快许多，因为后者要使用许多的 CPU 时间来对文件多做一点压缩。我们的测试基于默认压缩级别。

传输文件的最后一步是验证复制过程没有损坏文件。可以使用许多方法，例如 *md5sum*，但再次对文件做完整扫描也相当昂贵。这也是压缩很有用的另外一个原因：压缩本身往往包括至少一个循环冗余检测（CRC），而它应该能发现任何错误，因此不需要做错误检测。

附录 D EXPLAIN

这个附录显示了如何调用“EXPLAIN”来获取关于查询执行计划的信息，以及如何解释输出。EXPLAIN 命令是查看查询优化器如何决定执行查询的主要方法。这个功能有局限性，并不总会说出真相，但它的输出是可以获取的最好信息，值得花时间了解，因为可以学习到查询是如何执行的。学会解释 EXPLAIN 将帮助你了解 MySQL 优化器是如何工作的。

调用 EXPLAIN

要使用 EXPLAIN，只需在查询中的 SELECT 关键字之前增加 EXPLAIN 这个词。MySQL 会在查询上设置一个标记。当执行查询时，这个标记会使其返回关于在执行计划中每一步的信息，而不是执行它。它会返回一行或多行信息，显示出执行计划中的每一部分和执行的次序。

下面是一个可能的最简单的 EXPLAIN 结果。

```
mysql> EXPLAIN SELECT 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra: No tables used
```

在查询中每个表在输出中只有一行。如果查询是两个表的联接，那么输出中将有两行。别名表单算为一个表，因此，如果把一个表与自己联接，输出中也会有两行。“表”的意义在这里相当广：可以是一个子查询，一个 UNION 结果，等等。稍后会看到为什么是这样。

720 EXPLAIN有两个主要的变种。

- EXPLAIN EXTENDED看起来和正常的EXPLAIN的行为一样，但它会告诉服务器"逆向编译"执行计划为一个SELECT语句。可以通过紧接其后运行SHOW WARNINGS看到这个生成的语句。这个语句直接来自执行计划，而不是原SQL语句，到这点上已经变成一个数据结构。在大部分场景下它都与原语句不相同。你可以检测查询优化器到底是如何转化语句的。EXPLAIN EXTENDED在MySQL 5.0和更新版本中可用，在MySQL 5.1（稍后会做更多讨论）额外增加了一个filtered列。
- EXPLAIN PARTITIONS会显示查询将访问的分区，如果查询是基于分区表的话。它只在MySQL 5.1和更新版本中存在。

认为增加EXPLAIN时MySQL不会执行查询，这是一个常见的错误。事实上，如果查询在FROM子句中包括子查询，那么MySQL实际上会执行子查询，将其结果放在一个临时表中，然后完成外层查询优化。它必须在可以完成外层查询优化之前处理所有类似的子查询，这对于EXPLAIN来说是必须要做的[注1]。这意味着如果语句包含开销较大的子查询或使用临时表算法的视图，实际上会给服务器带来大量工作。

要意识到EXPLAIN只是个近似结果，别无其他。有时候它是一个很好的近似，但在其他时候，可能与真相相差甚远。以下是一些相关的限制。

- EXPLAIN根本不会告诉你触发器、存储过程或UDF会如何影响查询。
- 它并不支持存储过程，尽管可以手动抽取查询并单独地对其进行EXPLAIN操作。
- 它并不会告诉你MySQL在查询执行中所做的特定优化。
- 它并不会显示关于查询的执行计划的所有信息（MySQL开发者会尽可能增加更多信息）。
- 它并不区分具有相同名字的事物。例如，它对内存排序和临时文件都使用"filesort"，并且对于磁盘上和内存中的临时表都显示"Using temporary"。
- 721 可能会误导。例如，它会对一个有着很小LIMIT的查询显示全索引扫描。（MySQL 5.1的EXPLAIN关于检查的行数会显示更精确的信息，但早期版本并不考虑LIMIT。）

重写非SELECT查询

MySQL EXPLAIN只能解释SELECT查询，并不会对存储程序调用和INSERT、UPDATE、DELETE或其他语句做解释。然而，你可以重写某些非SELECT查询以利用EXPLAIN。为了达到这个目的，只需要将该语句转化成一个等价的访问所有相同列的SELECT。任何提及的列都必须在SELECT列表，关联子句，或者WHERE子句中。

注1：这个限制在MySQL 5.6中将被取消。

例如，假如你想重写下面的 UPDATE 语句以使其可以利用 EXPLAIN。

```
UPDATE sakila.actor
   INNER JOIN sakila.film_actor USING (actor_id)
SET actor.last_update=film_actor.last_update;
```

下面的 EXPLAIN 语句并不等价于上面的 UPDATE，因为它并不要求服务器从任何一个表上获取 last_update 列。

```
mysql> EXPLAIN SELECT film_actor.actor_id
    -> FROM sakila.actor
    ->    INNER JOIN sakila.film_actor USING (actor_id)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: index
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: NULL
         rows: 200
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.actor.actor_id
         rows: 13
        Extra: Using index
```

这个差别非常重要。例如，输出结果显示 MySQL 将使用覆盖索引，但是，当检索并更新 last_updated 列时，就无法使用覆盖索引了。下面这种改写法就更接近原来的语句：

722

```
mysql> EXPLAIN SELECT film_actor.last_update, actor.last_update
    -> FROM sakila.actor
    ->    INNER JOIN sakila.film_actor USING (actor_id)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 200
        Extra:
*************************** 2. row ***************************
           id: 1
```

```
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.actor.actor_id
         rows: 13
        Extra:
```

像这样重写查询并不非常科学，但对帮助理解查询是怎么做的经常已足够好了。[注2]

显示计划时，对于写查询并没有"等价"的读查询，理解这一点非常重要。一个 SELECT 查询只需要找到数据的一份副本并返回。而任何修改数据的查询必须在所有索引上查找并修改其所有副本。这常常比看起来等价的 SELECT 查询的消耗要高得多。

EXPLAIN 中的列

EXPLAIN 的输出总是有相同的列（只有 EXPLAIN EXTENDED 在 MySQL 5.1 中增加了一个 filtered 列，EXPLAIN PARTITIONS 增加了一个 Partitions 列）。可变的是行数及内容。然而，为了保持我们的例子简洁明了，我们在本附录中不总是显示所有的列。

在接下来的小节中，我们将展示在 EXPLAIN 结果中每一列的意义。记住，输出中的行以 MySQL 实际执行的查询部分的顺序出现，而这个顺序不总是与其在原始 SQL 中的相一致。

723

id 列

这一列总是包含一个编号，标识 SELECT 所属的行。如果在语句当中没有子查询或联合，那么只会有唯一的 SELECT，于是每一行在这个列中都将显示一个 1。否则，内层的 SELECT 语句一般会顺序编号，对应于其在原始语句中的位置。

MySQL 将 SELECT 查询分为简单和复杂类型，复杂类型可分成三大类：简单子查询、所谓的派生表（在 FROM 子句中的子查询）[注3]，以及 UNION 查询。下面是一个简单的子查询。

```
mysql> EXPLAIN SELECT (SELECT 1 FROM sakila.actor LIMIT 1) FROM sakila.film;
+----+-------------+-------+...
| id | select_type | table |...
+----+-------------+-------+...
|  1 | PRIMARY     | film  |...
|  2 | SUBQUERY    | actor |...
+----+-------------+-------+...
```

注 2： MySQL 5.6 将允许解释非 SELECT 查询。万岁！

注 3： FROM 子句中的子查询是派生表"这一表述是对的，但"派生表是 FROM 子句中的子查询"则不对，术语"派生表"在 SQL 中含义很宽泛。

FROM 子句中的子查询和联合给 id 列增加了更多复杂性。下面是一个 FROM 子句中的基本子查询。

```
mysql> EXPLAIN SELECT film_id FROM (SELECT film_id FROM sakila.film) AS der;
+----+-------------+------------+...
| id | select_type | table      |...
+----+-------------+------------+...
|  1 | PRIMARY     | <derived2> |...
|  2 | DERIVED     | film       |...
+----+-------------+------------+...
```

如你所知，这个查询执行时有一个匿名临时表。MySQL 内部通过别名（der）在外层查询中引用这个临时表，在更复杂的查询中可以看到 ref 列。

最后，下面是一个 UNION 查询。

```
mysql> EXPLAIN SELECT 1 UNION ALL SELECT 1;
+------+--------------+------------+...
| id   | select_type  | table      |...
+------+--------------+------------+...
|  1   | PRIMARY      | NULL       |...
|  2   | UNION        | NULL       |...
| NULL | UNION RESULT | <union1,2> |...
+------+--------------+------------+...
```

注意 UNION 结果输出中的额外行。UNION 结果总是放在一个匿名临时表中，之后 MySQL 将结果读取到临时表外。临时表并不在原 SQL 中出现，因此它的 id 列是 NULL。与之前的例子相比（演示子查询的那个 FROM 子句中），从这个查询产生的临时表在结果中出现在最后一行，而不是第一行。 724

到目前为止这些都非常直截了当，但这三类语句的混合则会使输出变得非常复杂，我们稍后就会看到。

select_type 列

这一列显示了对应行是简单还是复杂 SELECT（如果是后者，那么是三种复杂类型中的哪一种）。SIMPLE 值意味着查询不包括子查询和 UNION。如果查询有任何复杂的子部分，则最外层部分标记为 PRIMARY，其他部分标记如下。

SUBQUERY

包含在 SELECT 列表中的子查询中的 SELECT（换句话说，不在 FROM 子句中）标记为 SUBQUERY。

DERIVED

DERIVED 值用来表示包含在 FROM 子句的子查询中的 SELECT，MySQL 会递归执行并将结果放到一个临时表中。服务器内部称其“派生表”，因为该临时表是从子查询中派生来的。

UNION

在 UNION 中的第二个和随后的 SELECT 被标记为 UNION。第一个 SELECT 被标记就好像它以部分外查询来执行。这就是之前的例子中在 UNION 中的第一个 SELECT 显示为 PRIMARY 的原因。如果 UNION 被 FROM 子句中的子查询包含，那么它的第一个 SELECT 会被标记为 DERIVED。

UNION RESULT

用来从 UNION 的匿名临时表检索结果的 SELECT 被标记为 UNION RESULT。

除了这些值，SUBQUERY 和 UNION 还可以被标记为 DEPENDENT 和 UNCACHEABLE。DEPENDENT 意味着 SELECT 依赖于外层查询中发现的数据；UNCACHEABLE 意味着 SELECT 中的某些特性阻止结果被缓存于一个 Item_cache 中。（Item_cache 未被文档记载；它与查询缓存不是一回事，尽管它可以被一些相同类型的构件否定，例如 RAND() 函数。）

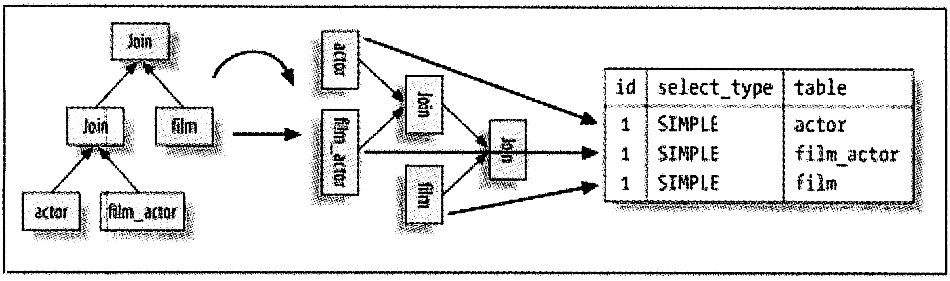
table 列

这一列显示了对应行正在访问哪个表。在通常情况下，它相当明了：它就是那个表，或是该表的别名（如果 SQL 中定义了别名）。

725 可以在这一列中从上往下观察 MySQL 的关联优化器为查询选择的关联顺序。例如，可以看到在下面的查询中 MySQL 选择的关联顺序不同于语句中所指定的顺序。

```
mysql> EXPLAIN SELECT film.film_id
    -> FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    ->    INNER JOIN sakila.actor USING(actor_id);
+----+-------------+------------+...
| id | select_type | table      |...
+----+-------------+------------+...
|  1 | SIMPLE      | actor      |...
|  1 | SIMPLE      | film_actor |...
|  1 | SIMPLE      | film       |...
+----+-------------+------------+...
```

想起我们在第 6 章中展示的左侧深度优先（left-deep）树了吗？ MySQL 的查询执行计划总是左侧深度优先树。如果把这个计划放倒，就能按顺序读出叶子节点，它们直接对应于 EXPLAIN 中的行。之前的查询计划看起来如图 D-1 所示。



图D-1：查询执行计划与EXPLAIN中的行相对应的方式

派生表和联合

当 FROM 子句中有子查询或有 UNION 时，table 列会变得复杂得多。在这些场景下，确实没有一个“表”可以参考到，因为 MySQL 创建的匿名临时表仅在查询执行过程中存在。

当在 FROM 子句中有子查询时，table 列是 <derived*N*> 的形式，其中 *N* 是子查询的 id。这总是“向前引用”——换言之，*N* 指向 EXPLAIN 输出中后面的一行。

当有 UNION 时，UNION RESULT 的 table 列包含一个参与 UNION 的 id 列表。这总是“向后引用”，因为 UNION RESULT 出现在 UNION 中所有参与行之后。如果在列表中有超过 20 个 id，table 列可能被截断以防止太长，此时不可能看到所有的值。幸运的是，仍然可以推测包括哪些行，因为你可以看到第一行的 id。在这一行和 UNION RESULT 之间出现的一切都会以某种方式被包含。

一个复杂 SELECT 类型的例子 726

下面是一个无意义的查询，我们这里把它用作某种复杂 SELECT 类型的紧凑示例。

```
EXPLAIN
SELECT actor_id,
   (SELECT 1 FROM sakila.film_actor WHERE film_actor.actor_id =
      der_1.actor_id LIMIT 1)
FROM (
   SELECT actor_id
   FROM sakila.actor LIMIT 5
) AS der_1
UNION ALL
SELECT film_id,
   (SELECT @var1 FROM sakila.rental LIMIT 1)
FROM (
   SELECT film_id,
      (SELECT 1 FROM sakila.store LIMIT 1)
   FROM sakila.film LIMIT 5
) AS der_2;
```

LIMIT 子句只是为了方便起见，以防你打算不以 EXPLAIN 方式执行来看结果。下面是 EXPLAIN 的结果。

```
+------+----------------------+------------+...
| id   | select_type          | table      |...
+------+----------------------+------------+...
|  1   | PRIMARY              | <derived3> |...
|  3   | DERIVED              | actor      |...
|  2   | DEPENDENT SUBQUERY   | film_actor |...
|  4   | UNION                | <derived6> |...
|  6   | DERIVED              | film       |...
|  7   | SUBQUERY             | store      |...
|  5   | UNCACHEABLE SUBQUERY | rental     |...
| NULL | UNION RESULT         | <union1,4> |...
+------+----------------------+------------+...
```

我们特意让每个查询部分访问不同的表，以便可以弄清问题所在，但仍然难以解决！从最上面开始看。

- 第 1 行向前引用了 der_1，这个查询被标记为 <derived3>。在原 SQL 中是第 2 行。想了解输出中哪些行引用了 <derived3> 中的 SELECT 语句，往下看……
- ……第 2 行，它的 id 是 3。因为它是查询中第 3 个 SELECT 的一部分，归为 DERIVED 类型是因为它嵌套在 FROM 子句中的子查询内部。在原 SQL 中为第 6 ~ 7 行。
- 727 第 3 行的 id 为 2。在原 SQL 中为第 3 行。注意，它在具有更高 id 的行后面，暗示后面再执行，这是合理的。它被归为 DEPENDENT SUBQUERY，意味着其结果依赖于外层查询（亦即某个相关子查询）。本例中的外查询是从第 2 行开始，从 der_1 中检索数据的 SELECT。
- 第 4 行被归为 UNION，意味着它是 UNION 中的第 2 个或之后的 SELECT。它的表为 <derived6>，意味着是从子句 FROM 的子查询中检索数据并附加到 UNION 的临时表。像之前一样，要找到显示这个子查询的查询计划的 EXPLAIN 行，必须往下看。
- 第 5 行是在原 SQL 中第 13、14 和 15 行定义的 der_2 子查询，EXPLAIN 称其为 <derived6>。
- 第 6 行是 <derived6> 的 SELECT 列表中的一个普通子查询，它的 id 为 7，这非常重要……
- ……因为它比 5 大，而 5 是第 7 行的 id。为什么重要？因为它显示了 <derived6> 子查询的边界。当 EXPLAIN 输出 SELECT 类型为 DERIVED 的一行时，表示一个“嵌套范围”开始。如果后续行的 id 更小（本例中，5 小于 6），意味着嵌套范围已经被关闭。这就让我们知道第 7 行是从 <derived6> 中检索数据的 SELECT 列表中的部分——例如，第 4 行的 SELECT 列表的一部分（原 SQL 中第 11 行）。这个例子相当容易理解，不需要知道嵌套范围的意义和规则，当然有时候并不是这么容易。关于输出中的这一行另外一个要注意的是，因为有用户变量，它被列为 UNCACHEABLE SUBQUERY。

- 最后一行是 UNION RESULT。它代表从 UNION 的临时表中读取行的阶段。你可以从这行开始反过来向后，如果你愿意的话。它会返回 id 是 1 和 4 的行结果，它们分别引用了 <derived3> 和 <derived6>。

如你所见，这些复杂的 SELECT 类型的组合会使 EXPLAIN 的输出相当难懂。理解规则会使其简单些，但仍然需要多实践。

阅读 EXPLAIN 的输出经常需要在列表中跳来跳去。例如，再查看第一行输出。仅仅盯着看，是无法知道它是 UNION 的一部分的。只有看到最后一行你才会明白过来。

type 列

MySQL 用户手册上说这一列显示了“关联类型”，但我们认为更准确的说法是访问类型——换言之就是 MySQL 决定如何查找表中的行。下面是最重要的访问方法，依次从最差到最优。

ALL 728

这就是人们所称的全表扫描，通常意味着 MySQL 必须扫描整张表，从头到尾，去找到需要的行。（这里也有个例外，例如在查询里使用了 LIMIT，或者在 Extra 列中显示“Using distinct/not exists”。）

index

这个跟全表扫描一样，只是 MySQL 扫描表时按索引次序进行而不是行。它的主要优点是避免了排序；最大的缺点是要承担按索引次序读取整个表的开销。这通常意味着若是按随机次序访问行，开销将会非常大。

如果在 Extra 列中看到“Using index”，说明 MySQL 正在使用覆盖索引，它只扫描索引的数据，而不是按索引次序的每一行。它比按索引次序全表扫描的开销要少很多。

range

范围扫描就是一个有限制的索引扫描，它开始于索引里的某一点，返回匹配这个值域的行。这比全索引扫描好一些，因为它用不着遍历全部索引。显而易见的范围扫描是带有 BETWEEN 或在 WHERE 子句里带有 > 的查询。

当 MySQL 使用索引去查找一系列值时，例如 IN() 和 OR 列表，也会显示为范围扫描。然而，这两者其实是相当不同的访问类型，在性能上有重要的差异。更多信息可以查看第 5 章的文章“什么是范围条件”。

此类扫描的开销跟索引类型相当。

ref

这是一种索引访问（有时也叫做索引查找），它返回所有匹配某个单个值的行。然而，它可能会找到多个符合条件的行，因此，它是查找和扫描的混合体。此类索引

访问只有当使用非唯一性索引或者唯一性索引的非唯一性前缀时才会发生。把它叫做 ref 是因为索引要跟某个参考值相比较。这个参考值或者是一个常数，或者是来自多表查询前一个表里的结果值。

ref_or_null 是 ref 之上的一个变体，它意味着 MySQL 必须在初次查找的结果里进行第二次查找以找出 NULL 条目。

eq_ref

使用这种索引查找，MySQL 知道最多只返回一条符合条件的记录。这种访问方法可以在 MySQL 使用主键或者唯一性索引查找时看到，它会将它们与某个参考值做比较。MySQL 对于这类访问类型的优化做得非常好，因为它知道无须估计匹配行的范围或在找到匹配行后再继续查找。

const, system

当 MySQL 能对查询的某部分进行优化并将其转换成一个常量时，它就会使用这些
729 访问类型。举例来说，如果你通过将某一行的主键放入 WHERE 子句里的方式来选取此行的主键，MySQL 就能把这个查询转换为一个常量。然后就可以高效地将表从联接执行中移除。

NULL

这种访问方式意味着 MySQL 能在优化阶段分解查询语句，在执行阶段甚至用不着再访问表或者索引。例如，从一个索引列里选取最小值可以通过单独查找索引来完成，不需要在执行时访问表。

possible_keys 列

这一列显示了查询可以使用哪些索引，这是基于查询访问的列和使用的比较操作符来判断的。这个列表是在优化过程的早期创建的，因此有些罗列出来的索引可能对于后续优化过程是没用的。

key 列

这一列显示了 MySQL 决定采用哪个索引来优化对该表的访问。如果该索引没有出现在 possible_keys 列中，那么 MySQL 选用它是出于另外的原因——例如，它可能选择了一个覆盖索引，哪怕没有 WHERE 子句。

换句话说，possible_keys 揭示了哪一个索引能有助于高效地行查找，而 key 显示的是优化采用哪一个索引可以最小化查询成本（更多详情请参阅第 6 章中关于优化的成本度量值）。下面就是一个例子。

```
mysql> EXPLAIN SELECT actor_id, film_id FROM sakila.film_actor\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: index
possible_keys: NULL
          key: idx_fk_film_id
      key_len: 2
          ref: NULL
         rows: 5143
        Extra: Using index
```

key_len 列

该列显示了 MySQL 在索引里使用的字节数。如果 MySQL 正在使用的只是索引里的某些列，那么就可以用这个值来算出具体是哪些列。要记住，MySQL 5.5 及之前版本只能使用索引的最左前缀。举例来说，sakila.film_actor 的主键是两个 SMALLINT 列，并且每个 SMALLINT 列是两字节，那么索引中的每项是 4 字节。以下就是一个查询的示例：

```
mysql> EXPLAIN SELECT actor_id, film_id FROM sakila.film_actor WHERE actor_id=4;
...+------+--------------+---------+---------+...
...| type | possible_keys | key     | key_len |...
...+------+--------------+---------+---------+...
...| ref  | PRIMARY       | PRIMARY | 2       |...
...+------+--------------+---------+---------+...
```
730

基于结果中的 key_len 列，可以推断出查询使用唯一的首列——actor_id 列，来执行索引查找。当我们计算列的使用情况时，务必把字符列中的字符集也考虑进去。

```
mysql> CREATE TABLE t (
    ->    a char(3) NOT NULL,
    ->    b int(11) NOT NULL,
    ->    c char(1) NOT NULL,
    ->    PRIMARY KEY  (a,b,c)
    -> ) ENGINE=MyISAM DEFAULT CHARSET=utf8 ;
mysql> INSERT INTO t(a, b, c)
    ->    SELECT DISTINCT LEFT(TABLE_SCHEMA, 3), ORD(TABLE_NAME),
    ->       LEFT(COLUMN_NAME, 1)
    ->    FROM INFORMATION_SCHEMA.COLUMNS:
mysql> EXPLAIN SELECT a FROM t WHERE a='sak' AND b = 112;
...+------+--------------+---------+---------+...
...| type | possible_keys | key     | key_len |...
...+------+--------------+---------+---------+...
...| ref  | PRIMARY       | PRIMARY | 13      |...
...+------+--------------+---------+---------+...
```

这个查询中平均长度为 13 字节，即为 a 列和 b 列的总长度。a 列是 3 个字符，utf8 下每一个最多为 3 字节，而 b 列是一个 4 字节整型。

MySQL 并不总显示一个索引真正使用了多少。例如，如果对一个前缀模式匹配执行 LIKE 查询，它会显示列的完全宽度正在被使用。

key_len 列显示了在索引字段中可能的最大长度，而不是表中数据使用的实际字节数。在前面例子中 MySQL 总是显示 13 字节，即使 a 列恰巧只包含一个字符长度。换言之，key_len 通过查找表的定义而被计算出，而不是表中的数据。

ref 列

这一列显示了之前的表在 key 列记录的索引中查找值所用的列或常量。下面是一个展示关联条件和别名组合的例子。注意，ref 列反映了在查询文本中 film 表是如何以 f 为别名的。

731

```
mysql> EXPLAIN
    -> SELECT STRAIGHT_JOIN f.film_id
    -> FROM sakila.film AS f
    ->    INNER JOIN sakila.film_actor AS fa
    ->       ON f.film_id=fa.film_id AND fa.actor_id = 1
    ->    INNER JOIN sakila.actor AS a USING(actor_id);
...+-------+...+--------------------+---------+------------------------+...
...| table |...| key                | key_len | ref                    |...
...+-------+...+--------------------+---------+------------------------+...
...| a     |...| PRIMARY            | 2       | const                  |...
...| f     |...| idx_fk_language_id | 1       | NULL                   |...
...| fa    |...| PRIMARY            | 4       | const,sakila.f.film_id |...
...+-------+...+--------------------+---------+------------------------+...
```

rows 列

这一列是 MySQL 估计为了找到所需的行而要读取的行数。这个数字是内嵌循环关联计划里的循环数目。也就是说它不是 MySQL 认为它最终要从表里读取出来的行数，而是 MySQL 为了找到符合查询的每一点上标准的那些行而必须读取的行的平均数。（这个标准包括 SQL 里给定的条件，以及来自联接次序上前一个表的当前列。）

根据表的统计信息和索引的选用情况，这个估算可能很不精确。在 MySQL 5.0 及更早的版本里，它也反映不出 LIMIT 子句。举例来说，下面这个查询不会真的检查 1 022 行。

```
mysql> EXPLAIN SELECT * FROM sakila.film LIMIT 1\G
...
         rows: 1022
```

通过把所有 rows 列的值相乘，可以粗略地估算出整个查询会检查的行数。例如，以下这个查询大约会检查 2 600 行。

```
mysql> EXPLAIN
    -> SELECT f.film_id
    -> FROM sakila.film AS f
    ->    INNER JOIN sakila.film_actor AS fa USING(film_id)
    ->    INNER JOIN sakila.actor AS a USING(actor_id);
...+------+...
...| rows |...
...+------+...
...|  200 |...
...|   13 |...
...|    1 |...
...+------+...
```

要记住这个数字是 MySQL 认为它要检查的行数，而不是结果集里的行数。同时也要认识到有很多优化手段，例如关联缓冲区和缓存，无法影响到行数的显示。MySQL 可能不必真的读所有它估计到的行，它也不知道任何关于操作系统或硬件缓存的信息。 732

filtered 列

这一列是在 MySQL 5.1 里新加进去的，在使用 EXPLAIN EXTENDED 时出现。它显示的是针对表里符合某个条件（WHERE 子句或联接条件）的记录数的百分比所做的一个悲观估算。如果你把 rows 列和这个百分比相乘，就能看到 MySQL 估算它将和查询计划里前一个表关联的行数。在写作本书的时候，优化器只有在使用 ALL、index、range 和 index_merge 访问方法时才会用这一估算。

为了说明这一列的输出形式，我们创建了下面这样一张表。

```
CREATE TABLE t1 (
   id INT NOT NULL AUTO_INCREMENT,
   filler char(200),
   PRIMARY KEY(id)
);
```

然后，我们往表里插入 1000 行记录，并在 filler 列里随机填充一些文字。它的用途是防止 MySQL 在我们将要运行的查询里使用覆盖索引。

```
mysql> EXPLAIN EXTENDED SELECT * FROM t1 WHERE id < 500\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1000
     filtered: 49.40
        Extra: Using where
```

MySQL 可以使用范围访问从表里获取到所有 ID 不超过 500 的行，但是，它没这么做，这是因为那样只能去除大约一半的记录，它认为全表扫描也不是太昂贵。因此，它使用了全表扫描和 WHERE 子句来过滤输出行。它知道使用 WHERE 子句可以从结果里过滤掉多少条记录，因为范围访问的成本是可以估算出来的。这也就是 49.40% 出现在 filtered 列上的原因。

Extra 列

这一列包含的是不适合在其他列显示的额外信息。MySQL 用户手册里记录了大多数可以在这里出现的值。其中许多在本书中已经提到过。

733 常见的最重要的值如下。

"*Using index*"

　　此值表示 MySQL 将使用覆盖索引，以避免访问表。不要把覆盖索引和 index 访问类型弄混了。

"*Using where*"

　　这意味着 MySQL 服务器将在存储引擎检索行后再进行过滤。许多 WHERE 条件里涉及索引中的列，当（并且如果）它读取索引时，就能被存储引擎检验，因此不是所有带 WHERE 子句的查询都会显示 "Using where"。有时 "Using where" 的出现就是一个暗示：查询可受益于不同的索引。

"*Using temporary*"

　　这意味着 MySQL 在对查询结果排序时会使用一个临时表。

"*Using filesort*"

　　这意味着 MySQL 会对结果使用一个外部索引排序，而不是按索引次序从表里读取行。MySQL 有两种文件排序算法，你可以在第 6 章读到相关内容。两种方式都可以在内存或磁盘上完成。EXPLAIN 不会告诉你 MySQL 将使用哪一种文件排序，也不会告诉你排序会在内存里还是磁盘上完成。

"*Range checked for each record (index map: N)*"

　　这个值意味着没有好用的索引，新的索引将在联接的每一行上重新估算。*N* 是显示在 possible_keys 列中索引的位图，并且是冗余的。

树形格式的输出

MySQL 用户往往更希望把 EXPLAIN 的输出格式化成一棵树，更加精确地展示执行计划。实际上，EXPLAIN 查看查询计划的方式确实有点笨拙，树状结构也不适合表格化的输出。

当 Extra 列里有大量的值时，缺点更明显，使用 UNION 也是这样。UNION 跟 MySQL 能做的其他类型的联接不太一样，它不太适合 EXPLAIN。

如果对 EXPLAIN 的规则和特性有充分的了解，使用树形结构的执行计划也是可行的。但是这有点枯燥，最好还是留给自动化的工具处理。Percona Toolkit 包含了 *pt-visual-explain*，它就是这样一个工具。

MySQL 5.6 中的改进 734

MySQL 5.6 中将包括一个对 EXPLAIN 的重要改进：能对类似 UPDATE、INSERT 等的查询进行解释。尽管可以将 DML 语句转化为准等价的"SELECT"查询并 EXPLAIN，但结果并不会完全反映语句是如何执行的，因而这仍然非常有帮助。在开发使用类似 Percona Toolkit 中的 *pt-upgrade* 时曾尝试使用过那个技术，我们不止一次发现，在将查询转化为 SELECT 时，优化器并不能按我们预期的代码路径执行。因而，EXPLAIN 一个查询而不需要转化为 SELECT，对我们理解执行过程中到底发生什么，是非常有帮助的。

MySQL 5.6 还将包括对查询优化和执行引擎的一系列改进，允许匿名的临时表尽可能晚地被具体化，而不总是在优化和执行使用到此临时表的部分查询时创建并填充它们。这将允许 MySQL 可以直接解释带子查询的查询语句，而不需要先实际地执行子查询。

最后，MySQL 5.6 将通过在服务器中增加优化跟踪功能的方式改进优化器的相关部分。这将允许用户查看优化器做出的抉择，以及输入（例如，索引的基数）和抉择的原因。这非常有帮助，不仅仅对理解服务器选择的执行计划如此，对为什么选择这个计划也如此。

附录 E 735

锁的调试

任何使用锁来控制资源共享的系统，锁的竞争问题都不好调试。当我们给某个表增加一列新字段，或者只是进行查询，就有可能发现其他请求锁住了操作的表或者行。此时，通常你所想做的事就是找出查询阻塞的原因，从而知道该杀死哪个进程。这个附录显示了如何达到这两个目标。

MySQL 服务器本身使用了几种类型的锁。如果查询正在等待一个服务器级别的锁，那么可以在 `SHOW PROCESSLIST` 的输出中看到蛛丝马迹。除了服务器级别的锁，任何支持行级别锁的存储引擎，例如 InnoDB，都实现了自己的锁。在 MySQL 5.0 和更早版本中，服务器层无法主动识别这些锁，它们往往对用户和数据库管理员不可见。在 MySQL 5.1 和后续版本中可见性有了提高。

服务器级别的锁等待

锁等待可能发生在服务器级别或存储引擎级别。[注1]（应用程序级别的锁可能也是一个问题，但我们在此只关注 MySQL。）下面是 MySQL 服务器使用的几种类型的锁。

表锁

表可以被显式的读锁和写锁进行锁定。这些锁有许多的变种，例如本地读锁。你可以在 MySQL 手册 `LOCK TABLES` 部分了解到这些变种。除了这些显式的锁外，查询过程中还有隐式的锁。

全局锁 736

可以通过 `FLUSH TABLES WITH READ LOCK` 或设置 `read_only=1` 来获取单个全局读锁。它与任何表锁都冲突。

注 1： 如果需要回忆关于服务器和存储引擎之间的隔离，请参考第 1 章中的图 1-1。

命名锁

命名锁是表锁的一种，服务器在重命名或删除一个表时创建。

字符锁

你可以用 GET_LOCK() 及其相关函数在服务器级别内锁住和释放任意一个字符串。

在接下来的章节中我们将更详细地查看每种类型的锁。

表锁

表锁既可以是显式的也可以是隐式的。显式的锁用 LOCK TABLES 创建。例如，如果在 *mysql* 会话中执行下列命令，将在 sakila.film 上获得一个显式的锁。

```
mysql> LOCK TABLES sakila.film READ;
```

如果再在另外一个会话中执行如下的命令，查询会挂起并且不会完成。

```
mysql> LOCK TABLES sakila.film WRITE;
```

你可以在第一个连接中看到等待线程。

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 7
   User: baron
   Host: localhost
     db: NULL
Command: Query
   Time: 0
  State: NULL
   Info: SHOW PROCESSLIST
*************************** 2. row ***************************
     Id: 11
   User: baron
   Host: localhost
     db: NULL
Command: Query
   Time: 4
  State: Locked
   Info: LOCK TABLES sakila.film WRITE
2 rows in set (0.01 sec)
```

737 可以注意到线程 11 的状态是 Locked。在 MySQL 服务器代码中只有一个线程会进入此状态：当一个线程持有该锁后，其他线程只能不断尝试获取。因而，如果看到这样的信息，你就知道线程在等待一个 MySQL 服务器中的锁，而不是存储引擎的。

然而，显式锁并不是阻塞这样一个操作的唯一类型的锁。我们前面也提到，服务器在查询过程中会隐式地锁住表。用一个长时间运行的查询可以很容易地展示这一点，长时间

查询可以通过 SLEEP() 函数轻松创建。

```
mysql> SELECT SLEEP(30) FROM sakila.film LIMIT 1;
```

当这个查询运行时，如果你再次尝试锁 sakila.film，操作会因隐式锁而挂起，就如同有显式锁一样。你会在进程列表中看到和之前一样的效果：

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 7
   User: baron
   Host: localhost
     db: NULL
Command: Query
   Time: 12
  State: Sending data
   Info: SELECT SLEEP(30) FROM sakila.film LIMIT 1
*************************** 2. row ***************************
     Id: 11
   User: baron
   Host: localhost
     db: NULL
Command: Query
   Time: 9
  State: Locked
   Info: LOCK TABLES sakila.film WRITE
```

在本例中，SELECT 查询的隐式读锁阻塞了 LOCK TABLES 中所请求的显式写锁。另外，隐式锁也会相互阻塞。

你可能想知道关于隐式锁和显式锁的差异。从内部来说，它们有相同的结构，由相同的 MySQL 服务器代码来控制。从外部来说，你可以通过 LOCK TABLES 和 UNLOCK TABLES 来控制显式锁。

然而，当涉及非 MyISAM 存储引擎时，它们之间有一个非常重要的区别。当创建显式锁时，它会按你的指令来做，但隐式锁就比较隐蔽并“有魔幻性”。服务器会在需要时自动地创建和释放隐式锁，并将它们传递给存储引擎。存储引擎感知到后，可能会“转换”这些锁。例如，InnoDB 有这样的相关规则：对一个给定的服务器级别的表锁，InnoDB 应该为其创建特定类型的 InnoDB 表锁。这也使得操作人很难理解 InnoDB 幕后到底做了什么。

找出谁持有锁 738

如果你看到许多的进程处于 Locked 状态，问题可能出在对 MyISAM 或者其他类似存储引擎的高并发访问。这会阻止你执行人工操作，例如给表增加索引等。如果一个 UPDATE 查询进入队列并等待 MyISAM 的表锁，此时就连 SELECT 也不会被允许运行。（关于 MySQL 锁队列和优先级，可以在 MySQL 用户手册中查到更多。）

在某些场景下，可以清楚地看到几个连接长时间持有某个锁，此时需要将它们杀死（或需要劝告用户不要阻挡这些连接的工作！）。但是如何找出那个连接呢？

目前没有 SQL 命令可以显示哪个线程持有阻塞你的查询的表锁。如果运行 SHOW PROCESSLIST，你会看到等待锁的进程，而不是哪个进程持有这些锁。幸运的是，有一个 *debug* 命令可以打印关于锁的信息到服务器的错误日志中，你可以使用 *mysqladmin* 工具来运行这个命令：

```
$ mysqladmin debug
```

在错误日志的输出中包括了许多的调试信息，在接近尾部可以看到像下面的一些信息。我们是这样创建这些输出的：在一个连接中锁住表，然后在另外一个连接中尝试再次对它加锁。

```
Thread database.table_name Locked/Waiting   Lock_type
7      sakila.film         Locked - read    Read lock  without concurrent inserts
8      sakila.film         Waiting - write  Highest priority write lock
```

可以看到线程 8 正在等待线程 7 持有的锁。

全局读锁

MySQL 服务器还实现了一个全局读锁，可以如下获取该锁。

```
mysql> FLUSH TABLES WITH READ LOCK;
```

如果此时在另外一个会话中尝试再锁这个表，结果会像之前一样挂起。

```
mysql> LOCK TABLES sakila.film WRITE;
```

如何判断这个查询正在等待全局读锁而不是一个表级别的锁？请看 SHOW PROCESSLIST 的输出。

739

```
mysql> SHOW PROCESSLIST\G
...
*************************** 2. row ***************************
     Id: 22
   User: baron
   Host: localhost
     db: NULL
Command: Query
   Time: 9
  State: Waiting for release of readlock
   Info: LOCK TABLES sakila.film WRITE
```

注意，查询的状态是 Waiting for release of readlock。这就是说查询正在等待一个全局读锁而不是表级别锁。

MySQL 没有提供查出谁持有全局读锁的方法。

命名锁

命名锁是一种表锁：服务器在重命名或删除一个表时创建。命名锁与普通的表锁相冲突，无论是隐式的还是显式的。例如，如果和之前一样使用 LOCK TABLES，然后在另外一个会话中尝试对此表重命名，查询会挂起，但这次不是处于 Locked 状态。

```
mysql> RENAME TABLE sakila.film2 TO sakila.film;
```

和前面一样，从进程列表找到获得锁的进程，其状态是 Waiting for table。

```
mysql> SHOW PROCESSLIST\G
...
*************************** 2. row ***************************
     Id: 27
   User: baron
   Host: localhost
     db: NULL
Command: Query
   Time: 3
  State: Waiting for table
   Info: rename table sakila.film to sakila.film 2
```

也可以在 SHOW OPEN TABLES 输出中看到命名锁的影响。

```
mysql> SHOW OPEN TABLES;
+----------+-----------+--------+-------------+
| Database | Table     | In_use | Name_locked |
+----------+-----------+--------+-------------+
| sakila   | film_text |      3 |           0 |
| sakila   | film      |      2 |           1 |
| sakila   | film2     |      1 |           1 |
+----------+-----------+--------+-------------+
3 rows in set (0.00 sec)
```

注意，两个名字（原名和新名）都被锁住了。sakila.film_text 因 sakila.film 上有个指向它的触发器而被锁，这也解释了另外一种锁方式，它们可以暗地里将自己放置到预期之外的地方。查询 sakila.film，触发器会使你悄悄地接触 sakila.film_text，因而隐式地锁住它。触发器实际不需要因重命名触发，确实如此，因此从技术上讲并不需要锁，但事实是：MySQL 的锁有时可能并不具有你所期望的细粒度。

740 MySQL 并没有提供任何一种方法来查明谁拥有命名锁，但这通常并不是问题，因为它们一般持有非常短的一段时间。当有冲突时，一般是由于命名锁在等待一个普通的表锁，而这通过先前展示的 *mysqladmin debug* 可以看到。

用户锁

在服务器中实现的最后一种锁是用户锁，它基本是一个命名互斥量。你需要指定锁的名称字符串，以及等待超时秒数。

```
mysql> SELECT GET_LOCK('my lock', 100);
+--------------------------+
| GET_LOCK('my lock', 100) |
+--------------------------+
|                        1 |
+--------------------------+
1 row in set (0.00 sec)
```

指令成功返回，这个线程就在命名互斥量上持有了一把锁。如果另外一个线程此时尝试锁相同的字符串，它将会挂起直到超时。这次进程列表显示了一个不同的进程状态。

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 22
   User: baron
   Host: localhost
     db: NULL
Command: Query
   Time: 9
  State: User lock
   Info: SELECT GET_LOCK('my lock', 100)
```

`User lock` 状态是这种类型的锁独有的。MySQL 没有提供查明谁拥有用户锁的方法。

InnoDB 中的锁等待

服务器级的锁要比存储引擎中的锁容易调试得多。各个存储引擎的锁互不相同，并且存储引擎可能不提供任何方法来查看内部的锁。本附录主要关注 InnoDB。

InnoDB 在 `SHOW INNODB STATUS` 的输出中显露了一些锁信息。如果事务正在等待某个锁，这个锁会显示在 `SHOW INNODB STATUS` 输出的 `TRANSACTIONS` 部分中。例如，如果在一个会话中执行下面的命令，将需要表中第一行的写锁。

```
mysql> SET AUTOCOMMIT=0;
mysql> BEGIN;
mysql> SELECT film_id FROM sakila.film LIMIT 1 FOR UPDATE;
```

如果在另外一个会话中运行相同的命令，查询将会因第一个会话中在那一行获取的锁而阻塞。可以在 SHOW INNODB STATUS 中看到影响（为了简洁起见我们对结果有所删减）。 741

```
LOCK WAIT 2 lock struct(s), heap size 1216
MySQL thread id 8, query id 89 localhost baron Sending data
SELECT film_id FROM sakila.film LIMIT 1 FOR UPDATE
------- TRX HAS BEEN WAITING 9 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 194 n bits 1072 index `idx_fk_language_id` of table
   `sakila/film` trx id 0 61714 lock_mode X waiting
```

最后一行显示查询在等待该表的 idx_fk_language_id 索引的 194 页上一个排他锁(lock_mode X)。最终，锁等待超时，查询返回一个错误。

```
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

不幸的是，由于看不到谁拥有锁，因此很难确定哪个事务导致这个问题。不过往往可以通过查看哪个事务打开了非常长的一段时间来有根据地猜测；还有另外一种方法，可以激活 InnoDB 锁监控器，它最多可以显示每个事务中拥有的 10 把锁。为了激活该监控器，需要在 InnoDB 存储引擎中创建一个特殊名字的表。[注2]

```
mysql> CREATE TABLE innodb_lock_monitor(a int) ENGINE=INNODB;
```

发起这个查询后，InnoDB 开始定时地（这个间隔时间可以变化，但通常是每分钟几次）打印 SHOW INNODB STATUS 的一个略微加强的版本的输出到标准输出中。在大多数系统中，这个输出被重定向到服务器的错误日志中；你可以检查它以查看哪个事务应该拥有那把锁。若想停掉锁监控器，删除这个表即可。

下面是锁监控器输出的相关例子。

```
1  ---TRANSACTION 0 61717, ACTIVE 3 sec, process no 5102, OS thread id 1141152080
2  3 lock struct(s), heap size 1216
3  MySQL thread id 11, query id 108 localhost baron
4  show innodb status
5  TABLE LOCK table `sakila/film` trx id 0 61717 lock mode IX
6  RECORD LOCKS space id 0 page no 194 n bits 1072 index `idx_fk_language_id` of tabl
   `sakila/film` trx id 0 61717 lock_mode X
7  Record lock, heap no 2 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
8   ... omitted ...
9
10 RECORD LOCKS space id 0 page no 231 n bits 168 index `PRIMARY` of table `sakila/fi
   trx id 0 61717 lock_mode X locks rec but not gap
11 Record lock, heap no 2 PHYSICAL RECORD: n_fields 15; compact format; info bits 0
12  ... omitted ...
```

请注意，第 3 行显示的 MySQL 线程 ID，跟进程列表里 ID 列的值是一样的。第 5 行显 742

注 2： InnoDB 把几个“神奇的”表名作为操作指令来用。当前采用的是动态可设置的服务器变量，但是，InnoDB 的方法已经使用了很长一段时间，所以仍然留有原先的行为方式。

示了该事务在表里有一个显式的独占表锁（IX）。第 6 ～ 8 行显示了索引里的锁。我们删除了第 8 行的信息，是因为它导出了这个锁定的记录，显得非常累赘。第 9 ～ 11 行显示了主键上相应的锁（FOR UPDATE 锁必须锁住整行，而不仅仅是索引）。

当锁监控器被激活的时候，SHOW INNODB STATUS 里也会有额外的信息，因此，实际上无须检查服务器的错误日志，就可以查看锁信息。

出于种种原因，锁监控器并不是最理想的。它的主要问题是锁信息非常冗长，因为导出了被锁定记录的十六进制格式和 ASCII 格式。它会填满错误日志，并且还会很轻易地溢出固定长度的 SHOW INNODB STATUS 输出结果。这意味着你可能无法查看到在那一段之后的其他输出信息。InnoDB 对每个事务打印锁的数量有硬编码限制，即每个事务只能打印出 10 个持有的锁，超过 10 个就无法输出，这意味着你可能看不到需要的锁信息。这还不算完，即使要找的东西确实在里面，也难以把它从所有锁的输出信息里定位出来。（只需在一个繁忙的系统上试一下，你就会体会到这一点。）

有两样东西能够使锁的输出信息更加有用。第一样是本书作者之一为 InnoDB 和 MySQL 服务器编写的一个补丁，包含在 Percona Server 和 MariaDB 中。这个补丁会移除输出结果里那些冗长的记录导出信息，默认会把锁信息包含到 SHOW INNODB STATUS 的输出中（因而锁监控器就无须激活了），还会增加动态可设置服务器变量来控制冗长的输出信息，以及每个事务能打印出的锁信息的个数。

第二个可选用的方法是使用 *innotop* 来解析和格式化输出结果。它的 Lock 模式能够显示锁信息，并通过连接和表优美地聚合在一起，因而能很快地看出哪一个事务持有指定表的锁。但是，这也并非是万无一失的方法，因为它是通过检查所有被锁定记录的导出信息来精确地找出那个被锁定记录。无论怎样，这还是要比常用的方法好很多，对于大多数用途都足够好了。

使用 INFORMATION_SCHEMA 表

使用 SHOW INNODB STATUS 来查看锁绝对是老派做法，现在 InnoDB 有 INFORMATION_SCHEMA 来显露它的事务和锁。

如果你看不到这个表，说明你使用的 InnoDB 版本还不够新。至少需要 MySQL 5.1 和 InnoDB 插件。如果你正在使用 MySQL 5.1，但没有看到 INNODB_LOCKS 表，请用 SHOW VARIABLES 检查 innodb_version 变量。如果没有看到这个变量，说明你还没有使用
743 InnoDB 插件，你需要它！如果看到了这个变量但没有那些表，那么你需要确保服务器配置文件的 plugin_load 设置中明确包括了那些表。详情请查阅 MySQL 用户手册。

幸运的是，MySQL 5.5 中不需要担心这些，InnoDB 的高级版本已经将它编译到服务器中。

对这些表可使用的查询，MySQL 和 InnoDB 手册都有样例，在此不再重复，但我们要增加两个自己的例子。例如，下面是一个显示谁阻塞和谁在等待，以及等待多久的查询。

```
SELECT r.trx_id AS waiting_trx_id,  r.trx_mysql_thread_id AS waiting_thread,
       TIMESTAMPDIFF(SECOND, r.trx_wait_started, CURRENT_TIMESTAMP) AS wait_time,
       r.trx_query AS waiting_query,
       l.lock_table AS waiting_table_lock,
       b.trx_id AS blocking_trx_id, b.trx_mysql_thread_id AS blocking_thread,
       SUBSTRING(p.host, 1, INSTR(p.host, ':') - 1) AS blocking_host,
       SUBSTRING(p.host, INSTR(p.host, ':') +1) AS blocking_port,
       IF(p.command = "Sleep", p.time, 0) AS idle_in_trx,
       b.trx_query AS blocking_query
FROM INFORMATION_SCHEMA.INNODB_LOCK_WAITS  AS w
INNER JOIN INFORMATION_SCHEMA.INNODB_TRX   AS b ON  b.trx_id = w.blocking_trx_id
INNER JOIN INFORMATION_SCHEMA.INNODB_TRX   AS r ON  r.trx_id = w.requesting_trx_id
INNER JOIN INFORMATION_SCHEMA.INNODB_LOCKS AS l ON  w.requested_lock_id = l.lock_id
LEFT JOIN  INFORMATION_SCHEMA.PROCESSLIST  AS p ON  p.id     = b.trx_mysql_thread_id
ORDER BY wait_time DESC\G
*************************** 1. row ***************************
    waiting_trx_id: 5D03
    waiting_thread: 3
         wait_time: 6
     waiting_query: select * from store limit 1 for update
waiting_table_lock: `sakila`.`store`
   blocking_trx_id: 5D02
   blocking_thread: 2
     blocking_host: localhost
     blocking_port: 40298
       idle_in_trx: 8
    blocking_query: NULL
```

603

结果显示线程 3 已经等待 store 表中的锁达 6s。它在线程 2 上被阻塞，而该线程已经空闲了 8s。

如果你因为线程在一个事务中空闲而正在遭受大量的锁操作，下面的这个变种查询可以告诉你有多少查询被哪些线程阻塞，而没有多余的无用信息。

```
SELECT CONCAT('thread ', b.trx_mysql_thread_id, ' from ', p.host) AS who_blocks,
       IF(p.command = "Sleep", p.time, 0) AS idle_in_trx,
       MAX(TIMESTAMPDIFF(SECOND, r.trx_wait_started, NOW())) AS max_wait_time,
       COUNT(*) AS num_waiters
FROM INFORMATION_SCHEMA.INNODB_LOCK_WAITS  AS w
INNER JOIN INFORMATION_SCHEMA.INNODB_TRX   AS b ON b.trx_id = w.blocking_trx_id
INNER JOIN INFORMATION_SCHEMA.INNODB_TRX   AS r ON r.trx_id = w.requesting_trx_id
LEFT JOIN  INFORMATION_SCHEMA.PROCESSLIST  AS p ON p.id     = b.trx_mysql_thread_id
GROUP BY who_blocks ORDER BY num_waiters DESC\G
*************************** 1. row ***************************
```

744

```
    who_blocks: thread 2 from localhost:40298
   idle_in_trx: 1016
 max_wait_time: 37
   num_waiters: 8
```

结果显示线程 2 已经空闲了更长的一段时间，并且至少有一个线程已经等待它释放它的锁长达 37s。有 8 个线程在等待线程 2 完成它的工作并提交。

我们发现 idle-in-transaction 锁操作是常见锁故障的一种起因，并且有时候很难诊断。Percona Toolkit 中的 *pt-kill* 可以配置用来杀死长时间运行的空闲事务以阻止这个场景。Percona Server 本身也支持一个空闲事务超时参数来完成相同的事情。

附录 F

在MySQL上使用Sphinx

Sphinx（*http://www.sphinxsearch.com*）是一个免费、开源的全文搜索引擎，设计着眼于与数据库完美结合。它有类似 DBMS 的特性，查询速度非常快，支持分布式检索，并且可扩展性好。它可以高效利用内存和磁盘 I/O，缓解大型操作在这部分的瓶颈，这非常重要。

Sphinx 在 MySQL 上工作得很好。它可以被用来加速各种各样的查询，包括全文搜索。也可以用来在其他应用中执行快速的分组和排序操作。它遵从 MySQL 的通信协议，以及主要的 MySQL 的 SQL 语法，使用户就像操纵 MySQL 一样进行查询。Sphinx 对某些特定的查询非常有用：MySQL 的通用架构对真实世界中的大型数据库优化得并不好。简而言之，Sphinx 可以加强 MySQL 的功能和性能。

Sphinx 索引的源数据通常就是 MySQL SELECT 查询的结果，但是，也可以用不同类型的无限的数据来源来建立索引，每一个 Sphinx 示例都能搜索到无限的索引。举例来说，你可以从位于一台远程服务器上的 MySQL 实例拉几份文档放入索引里，从位于另一台远程服务器上的 PostgreSQL 实例拉几份文档过来，再加上几份本地的脚本通过 XML 管道机制输出的文档。

在本附录里，我们将列举一些能让 Sphinx 体现出性能增强的使用案例，然后讲述一下安装和配置 Sphinx 所需的主要步骤，接着详细说明它的功能特点，最后讨论几个现实中应用的例子。

746

一个典型的 Sphinx 搜索

我们用一个简单但是完整的 Sphinx 应用例子作为进一步讨论的起点。虽然 Sphinx 的 API 可用于多种编程语言，但是，在这里我们使用的是 PHP，因为它比较普及。

假设我们要实现的是一个用于比较购物引擎里的全文搜索，其具体需求如下。

- 对 MySQL 中的一个产品表维护一个可搜索的全文索引。
- 允许对产品的名称和描述进行全文搜索。
- 如有需要，能够用指定的分类缩小搜索范围。
- 不仅可以按关联度对搜索结果进行排序，也可以用物品的价格或提交日期来排序。

我们先在 Sphinx 配置文件里设置好数据源和索引。

```
source products
{
    type        = mysql
    sql_host    = localhost
    sql_user    = shopping
    sql_pass    = mysecretpassword
    sql_db      = shopping
    sql_query   = SELECT id, title, description, \
                  cat_id, price, UNIX_TIMESTAMP(added_date) AS added_ts \
                  FROM products
    sql_attr_uint        = cat_id
    sql_attr_float       = price
    sql_attr_timestamp   = added_ts
}

index products
{
    source     = products
    path       = /usr/local/sphinx/var/data/products
    docinfo    = extern
}
```

这个例子假设 MySQL 的 `shopping` 数据库中包含了 `products` 表，而此表中有供我们执行 SELECT 查询生成我们的 Sphinx 索引的列。该 Sphinx 索引也命名为 `products`。在创建新数据源和索引后，我们运行 *indexer* 程序来创建最初的全文索引数据文件，然后启动（或重启）*searchd* 后台进程以同步这些变更。

```
$ cd /usr/local/sphinx/bin
$ ./indexer products
$ ./searchd --stop
$ ./searchd
```

索引现在已经就绪可以用于查询了。我们用 Sphinx 捆绑的 *test.php* 样例脚本来测试它。

```
$ php -q test.php -i products ipod

Query 'ipod ' retrieved 3 of 3 matches in 0.010 sec.
Query stats:
    'ipod' found 3 times in 3 documents
Matches:
1. doc_id=123, weight=100, cat_id=100, price=159.99, added_ts=2008-01-03 22:38:26
2. doc_id=124, weight=100, cat_id=100, price=199.99, added_ts=2008-01-03 22:38:26
3. doc_id=125, weight=100, cat_id=100, price=249.99, added_ts=2008-01-03 22:38:26
```

最后一步是将搜索功能加到我们的网络应用中。我们需要基于用户输入设置排序和过滤选项，并让输出格式漂亮些。同时，因为 Sphinx 返回给客户端的只有文档的 ID 和配置属性——它没有存储任何原始文本数据——所以，我们还要从 MySQL 里读取对应的行数据。

```
<?php
include ( "sphinxapi.php" );
// ... other includes, MySQL connection code,
// displaying page header and search form, etc. all go here

// set query options based on end-user input
$cl = new SphinxClient ();
$sortby = $_REQUEST["sortby"];
if ( !in_array ( $sortby, array ( "price", "added_ts" ) ) )
    $sortby = "price";
if ( $_REQUEST["sortorder"]=="asc" )
    $cl->SetSortMode ( SPH_SORT_ATTR_ASC, $sortby );
else
    $cl->SetSortMode ( SPH_SORT_ATTR_DESC, $sortby );
$offset = ($_REQUEST["page"]-1)*$rows_per_page;
$cl->SetLimits ( $offset, $rows_per_page );

// issue the query, get the results
$res = $cl->Query ( $_REQUEST["query"], "products" );

// handle search errors
if ( !$res )
{
    print "<b>Search error:</b>" . $cl->GetLastError ();
    die;
}

// fetch additional columns from MySQL
$ids = join ( ",", array_keys ( $res["matches"] ) );
$r = mysql_query ( "SELECT id, title FROM products WHERE id IN ($ids)" )
    or die ( "MySQL error: " . mysql_error() );
while ( $row = mysql_fetch_assoc($r) )
{
    $id = $row["id"];
    $res["matches"][$id]["sql"] = $row;
}

// display the results in the order returned from Sphinx
$n = 1 + $offset;
foreach ( $res["matches"] as $id=>$match )
```

748
```
{
    printf ( "%d. <a href=details.php?id=%d>%s</a>, USD %.2f<br>\n",
        $n++, $id, $match["sql"]["title"], $match["attrs"]["price"] );
}

?>
```

尽管上面显示的这段代码看上去相当简单，但还是有些东西值得强调一下。

- SetLimits() 调用会告诉 Sphinx 只获取客户端要在页面上显示的行数。做这样的限定在 Sphinx 中很方便（不同于 MySQL 内建的搜索功能），不加限定的结果数目也可以通过 $result['total_found'] 来获得，而不需要任何额外开销。
- 因为 Sphinx 只索引 title 列，并没有存储它，因而必须从 MySQL 里读取数据。
- 我们使用一条单独的合成查询获取数据，把所有文档都放在 WHERE id IN (...) 子句中，而不是每个文档运行一次查询（这会非常低效）。
- 我们将从 MySQL 里获取到的行注入到全文搜索的结果集里，以保持原始的排列顺序。在下文里我们会对此做一些解释。
- 我们使用来自 Sphinx 和 MySQL 的数据来显示每一行。

那些由 PHP 写的行注入代码需要再做一些解释。我们不能简单地对 MySQL 查询的结果集做遍历，因为行的次序跟 WHERE id IN (...) 子句里指定的（多数情况下）不一样。但是，PHP 会对结果进行哈希（使用关联数组），保持匹配结果插入时的排列顺序，这样 Sphinx 就可以通过 $result["matches"] 返回排序正确的行了。因此，为了从 Sphinx 返回的匹配结果能保持正确的次序（而不是 MySQL 生成的那种半随机的次序），我们需要把 MySQL 的查询结果一个接一个地注入到 PHP 用来存储 Sphinx 匹配结果集的哈希中。

对于计数匹配和应用 LIMIT 子句，MySQL 和 Sphinx 在实现方式及性能上存在着比较大的区别。首先，LIMIT 在 Sphinx 里开销是比较低的。设想有一个 LIMIT 500,10 的子句，MySQL 会半随机地读出 510 行数据（这是比较慢的），然后丢弃掉其中的 500 行，而 Sphinx 会返回一组 ID，你可以用这些 ID 从 MySQL 上读取到实际所需的数据行。其次，Sphinx 总是返回指定的行数或者它在结果集里找到的实际匹配数目，而与 LIMIT 子句是怎么样的无关。 MySQL 无法做到这么高效，尽管在 MySQL 5.6 中对这个限制有部分的优化。

749 为什么要使用 Sphinx

Sphinx 可以在多个方面完善基于 MySQL 的应用程序，能补充 MySQL 的性能不足，还提供了 MySQL 所没有的功能。典型的使用场景如下。

- 快速、高效、可扩展和核心的全文搜索。
- 能在使用低选择性索引或无索引的列时优化 WHERE 条件。
- 优化 ORDER BY ... LIMIT *N* 查询以及 GROUP BY 查询。
- 并行地产生结果集。
- 向上扩展和向外扩展。
- 聚合分片数据。

我们在下面这些小节里对这些场景逐一进行探讨。然而，这个列表也不是完整的，Sphinx 的用户时不时还会发现新的应用方法。例如，Sphinx 最重要用途之一——快速扫描和过滤记录——就是由一位用户创造出来的，并不是 Sphinx 最初的设计目标之一。

高效、可扩展的全文搜索

MyISAM 的全文搜索能力对小数据集非常快，但随着数据量增长，性能会非常低。对百万级别的记录量和上 GB 的索引文本，查询时间会在 1 秒到超过 10 分钟之间变化，而这对于高性能的网站应用来说是不可接受的。尽管通过将数据分布到多个地方可能会扩展 MyISAM 的全文搜索，但这需要并行地运行查询并在应用程序中将结果合并，大大增加了中间层的复杂度。

Sphinx 运作速度要明显快于 MyISAM 内建的全文索引。比如说，它查询超过 1GB 的文本数据需要 10~100ms——最多可以扩展到每个 CPU 处理 10~100GB 的数据。Sphinx 还有如下优点。

- 它能对 InnoDB 及其他存储引擎里存储的数据进行索引，而不仅仅是 MyISAM。
- 它能对多个源表的混合数据创建索引，不限于单个表上的字段。
- 它能将来自多个索引的搜索结果进行动态整合。
- 除了能对文本列索引外，它的索引还可以包含无限数量的数字属性——跟“额外字段”一样。Sphinx 的属性可以是整型、浮点型和 UNIX 时间戳。
- 它能根据属性上的附加条件对全文搜索进行优化。
- 它的基于短语的排列算法能帮助它返回更多相关的结果。例如，如果你在一个歌词表中搜索“我爱你，亲爱的”，那么恰好包含该短语的歌曲将在最上面返回，之后才是那些只包含多次“爱”或“亲爱的”的歌曲。 750
- 它使得向外扩展更容易。

高效使用 WHERE 子句

有时你需要对很大的表（有几百万条记录）做 SELECT 查询，同时，几个 WHERE 条件里有索引选择性非常差（例如指定 WHERE 条件返回太多行）或者根本没有索引支持的字段。

常见的例子有：在一个社交网站上搜索用户，以及在一个拍卖网站上搜索物品。典型的搜索接口是让用户能在 WHERE 条件加 10 个或更多的列，而返回结果又是按其他列来排序。第 5 章中索引案例研究的例子，就是这样的一个应用，并且需要索引策略。

当有合适的数据结构和查询优化时，只要 WHERE 子句不包含太多的列，尚可以接受用 MySQL 来应付这些查询。但是，随着列的数目增加，支持所有可能搜索所需的索引数会呈指数级增长。单是要覆盖到四列的所有可能的组合情况，MySQL 就要达到极限了。它会变得非常慢，并且要花费很多系统开销去维护索引。这意味着对于许多 WHERE 条件，实际上不可能拥有它所需要的所有索引，你不得不在没有索引的条件下运行查询。

更重要的是，即使可以增加索引，也无法受益很多，除非它们具有良好的可选择性。有一个典型的例子是 gender 列，它几乎帮不上忙，因为会命中大约所有行中的一半。当索引因缺少可选择性而帮不上忙时，MySQL 一般会回到全表扫描。

Sphinx 运行这类查询的速度比 MySQL 快很多。你可以只将数据中所需要的列做成 Sphinx 索引。然后 Sphinx 会允许用两种方式来访问这些数据：用关键字索引搜索或全表扫描。在这两种方式里，Sphinx 都用到了过滤器，它相当于一个 WHERE 子句。但是，和 MySQL 不一样，MySQL 是在内部决定使用索引还是全扫描，而 Sphinx 是让你自己选择要使用哪一种访问方法。

要使用带筛选的全扫描，你可以指定一个空字符串用作搜索查询条件；要使用索引搜索，你可以在构建索引时加一些伪关键字进去，然后再搜索那些关键字。例如，如果你想搜索分类 123 里的物品，你可以在建索引时把“分类 123”关键字添加到文档里，然后针对“分类 123”做全文搜索。你可以使用 CONCAT() 函数把关键字加到已有的一个字段里，或者为了更好的灵活性，为这些伪关键字创建一个特别的全文搜索字段。通常而言，对于覆盖率超过 30% 无选择性值的行就应该选择筛选；而对于具有选择性的值覆盖面不超过 10% 的，应该使用伪关键字；如果目标值是处于 10%~30% 的灰色区域，就难说了。你应该做几次基准测试以找出最佳解决方案。

751 Sphinx 无论是执行索引搜索还是全扫描都快过 MySQL。有时，Sphinx 的全扫描实际上比 MySQL 的索引读取还要快。

找出结果集里的前几行

Web 应用常常需要用到结果集里按顺序排列的前 *N* 行。如同我们之前讨论过的，在 MySQL 5.5 和之前版本中很难优化。

最糟糕的情况就是根据 WHERE 条件找到了许多行（假设有 100 万行），而 ORDER BY 里的列却没有被索引过。MySQL 使用索引识别出所有匹配的行，然后使用半随机磁盘读，把

记录一行接一行地读到排序缓冲区里，接着用一种文件排序将这些结果进行排序，最后丢弃其中的绝大多数。它会临时存储和处理整个结果集，忽略 LIMIT 子句，搅乱 RAM。如果排序缓冲区放不下整个结果集，还需要用到临时表，引发更多的磁盘 I/O。

这里还有一个极端的例子，你可能会认为这在真实世界里几乎不会发生，但事实上，它常常会发生。MySQL 在用于排序的索引方面是有限制的——只使用索引的最左边部分，不支持松散索引扫描（loose index scan），并且只允许一个单独的范围条件——这意味着真实世界里的查询不能从这些索引中受益。即使能够受益，使用半随机磁盘 I/O 来获取行也是一个性能杀手。

要对结果集进行分页，常常需要做形如 SELECT ... LIMIT *N*,*M* 的查询，这是 MySQL 的另一个性能问题。它们会从磁盘里读取 *N* + *M* 行，由此引发大量的随机 I/O，浪费内存资源。Sphinx 通过消除以下两个主要问题可以显著加速这类查询。

内存使用

Sphinx 对 RAM 的使用有严格限制，这个限制也是可以配置的。Sphinx 也支持与 MySQL LIMIT *N*, *M* 语法类似的结果集偏移量和大小，但是它还有一个 max_matches 选项。它以每个服务器和每个查询为基础，可控制类似“排序缓冲区”的大小。

I/O

如果属性是存储在 RAM 里的，Sphinx 就不会做任何 I/O 操作。即使属性是存储在磁盘上，Sphinx 也是通过顺序 I/O 来读取它们，这比 MySQL 使用半随机方式从磁盘获取行要快很多。

通过综合关联度（权重）、属性值和聚合函数值（当使用 GROUP BY 时），可以对搜索结果进行排序。排序子句的语法跟 SQL ORDER BY 子句类似。

```
<?php
$cl = new SphinxClient ();
$cl->SetSortMode ( SPH_SORT_EXTENDED, 'price DESC, @weight ASC' );
// more code and Query() call here...
?>
```

在本例中，price 是存储在索引中的用户指定的属性，@weight 是一个特殊的属性，在运 752
行时创建，它包含的是每一个搜索结果估算出来的关联度。你也可以使用算术表达式对结果再进行排序，算术表达式里可以包含属性值、常用数学运算符和函数。

```
<?php
$cl = new SphinxClient ();
$cl->SetSortMode ( SPH_SORT_EXPR, '@weight + log(pageviews)*1.5' );
// more code and Query() call here...
?>
```

优化 GROUP BY 查询

没有 GROUP BY 功能的话，对于日常的类 SQL 子句的支持也将不完整，因此，Sphinx 也支持 GPOUP BY。但与 MySQL 的通用实现不同，Sphinx 擅长高效筛选出 GROUP BY 任务所需要的实际子集。这个子集可以从如下场景拥有的大数据集（1 亿行）中生成报表：

- 结果是分组行中的少部分（这里的“少”是指 10 万～ 100 万量级）。
- 当从分布在集群中的多台机器上获取很多分组数据时，需要非常快的执行速度，同时近似的 COUNT(*) 结果可以接受。

这没有听起来那样严格。第一个场景实际上覆盖了所有可以想象的基于时间的报告。举个例子，每小时一次且持续时间为 10 年的一个详细报告，将会返回少于 90 000 行记录。第二个场景可以像这样通俗地表述：“尽可能迅速和精确地从 1 亿行的分片表中找出最重要的 20 条记录。”

这两种类型的查询会加速通用查询，但也可以在全文搜索应用中使用。许多应用不仅需要显示全文匹配结果，还要显示一些聚集结果。例如，许多搜索结果页显示了每个产品目录中有多少匹配的产品被找到，或一张按时间顺序的数量变化图。Sphinx 的 group-by 支持可以结合分组和全文搜索，消除在应用程序或 MySQL 中做分组的开销。

在 Sphinx 中的排序和分组都使用固定的内存，它的效率比类似数据集全部可以放在 RAM 中的 MySQL 查询要稍微（10% ～ 50%）高些。在本例中，大部分 Sphinx 的能力来源于它可以分散负载和大大减少延时。对于永不会全部放入 RAM 中的大数据集来说，可以使用内联属性（后面会定义）创建特别的基于磁盘的索引来生成报告。对这些索引
753 执行查询大约和读数据一样快——在现代硬件中大概是 30 ～ 100MB/s。在这个情况下，性能会比 MySQL 好许多倍，尽管结果是近似的。

与 MySQL 的 GROUP BY 最大的不同点是，在某些特定场景下 Sphinx 可能只生成近似结果。对于这点有两个原因。

- 使用固定量的内存来做分组。如果有太多的分组而不能放在 RAM 中，并且以某种“不幸的”顺序匹配，每组数量可能比实际值要小。
- 分布式搜索只发送联合结果，而不是一个节点一个节点进行匹配。如果在不同的节点上有重复记录，每组去重的数据可能会比实际值要大，因为可以去重的信息并没有在节点之间传送。

实践中，快速的近似 group-by 数量经常是可以接受的。如果这不可接受，往往也可能通过仔细地配置后台进程和客户端应用来取得精确结果。

你也可以产生与 COUNT(DISTINCT <*attribute*>) 等价的结果。例如，在一个拍卖网站上，你可以使用它来计算每个分类里卖家的精确数目。

最后，Sphinx 可以让你选取一个标准，然后用这个标准在每个分组里找到唯一的“最合适”的文档。例如，在以域分组且按每个域里的匹配数量排序结果集时，可以从每个域里选择相关度最高的文档。这在 MySQL 里不用复杂的查询是做不到的。

并行地取结果集

Sphinx 可以让你从相同数据中同时产生几份结果，同样是使用固定量的内存。作为对比，传统的 SQL 方法要么运行两个查询（并且希望两次运行中某些数据维持在缓存中），要么对每个搜索结果集创建一个临时表，Sphinx 的方法会产生显著的改进。

举例来说，假设你需要针对每天、每星期、每月定期生成该段时间周期里的报表，要用 MySQL 生成将不得不使用不同的 GROUP BY 子句运行三次查询，对源数据处理三次。然而，Sphinx 对于这些数据只要处理一次就行了，然后就可以并行地生成全部三份报表。

Sphinx 用一个 *multi-query* 机制来完成这项任务。不是一个接一个地发起查询，而是把几个查询做成一个批处理，然后在一个请求里提交。

```
<?php
$cl = new SphinxClient ();
$cl->SetSortMode ( SPH_SORT_EXTENDED, "price desc" );
$cl->AddQuery ( "ipod" );
$cl->SetGroupBy ( "category_id", SPH_GROUPBY_ATTR, "@count desc" );
$cl->AddQuery ( "ipod" );
$cl->RunQueries ();
?>
```

Sphinx 会分析这个查询，识别出可以合并的各查询部分，然后并行地执行这些查询。 754

例如，Sphinx 可能注意到，排序和分组的模式有些不同，而查询是相同的。这就是上面显示的示例代码里的情形——用 price 排序但用 category_id 分组。Sphinx 会创建几个排序队列来处理这些查询。当运行这些查询时，它会一次性地获取到行，然后把它们提交到所有的队列里。与一个接一个运行查询相比较，这个方法消除了几个冗余的全文检索或全扫描操作。

注意并行结果集的生成，虽然这是常见又重要的优化，但是，这只是更一般化的 multi-query 机制的一个特例。它不是唯一可能的优化。其中的指导法则是尽可能地把多个查询放在一个请求中，这通常有助于 Sphinx 开展内部优化操作。即使 Sphinx 无法并行处理这些查询，也可以节省网络往返。而且，如果将来 Sphinx 加入更多的优化功能，你的查询就能自动地使用到它们而无须做任何修改。

扩展

Sphinx 的可扩展性无论在水平方向（向外扩展）还是垂直方向（向上扩展）上都非常好。

Sphinx 完全可以在各机器之间分布。我们提到过的用户案例都能受益于跨 CPU 的分布工作。Sphinx 的搜索后台进程（*searchd*）支持特别的分布式索引，它知道哪些本地和远程的索引需要查询和聚合。这意味着向外扩展就是一次轻微的配置更改。你只需在各个节点间将数据分区，然后配置主节点使其能向其他节点并行地发起查询。这就是所有要做的事情。

你也能向上扩展，在单独的机器上增加更多 CPU 或内核，从而提高响应速度。为了达到这个目的，你可以在单台机器上运行好几个 *searchd* 实例，然后通过分布式索引，从另外的机器过来查询它们。另外一种可选择的方法是，你可以把一个单独的实例配置为能与它自己通信，这样并行的“远程”查询其实就发生在同一台机器上，但是使用的是不同的 CPU 或内核。

换句话说，使用 Sphinx 的单个查询也能使用到不止一个 CPU（多个并发查询会自动使用多个 CPU）。这跟 MySQL 有显著的区别，MySQL 的一个查询只能使用到一个 CPU，无论有多少个 CPU 可供使用。另外，Sphinx 在并发执行查询时不需要任何同步，这就避免了使用互斥体（一种同步机制）。而互斥体正是 MySQL 在多 CPU 环境下才会出现的声名狼藉的性能瓶颈之一。

向上扩展的另一个重要方面是扩展磁盘 I/O。不同的索引(包括部分更大型的分布式索引)能够轻松地放在不同的物理磁盘或 RAID 分卷上，以提高响应速度和吞吐量。这个方法
755 的一部分好处跟 MySQL 5.1 的分片表一样，后者能将数据分片存储到不同的位置上。不过，分布式索引也有一些比分片表更好的优点。Sphinx 使用分布式索引不仅可以分散负载，还能并行地处理一个查询的各个部分。相比之下，MySQL 的分片表能通过对分片的剪枝来优化一些查询（并不是所有的），但查询的处理是不能并行的。即使 Sphinx 和 MySQL 的分片都能提高查询的吞吐量，但如果查询是 I/O 密集的，则可以使用 Sphinx 让所有查询的响应速度得到线性的提高，而 MySQL 分片只能对那些可采用剪去整个分区的查询才能改善延时。

分布式搜索的工作流程非常直观。

1. 向所有远程服务器发出远程查询。
2. 执行连续的本地索引搜索。
3. 从每个远程服务器上读取部分搜索结果。
4. 将所有局部搜索结果合并成最终结果集，并将它返回给客户端。

如果硬件资源允许，也可以在同一台机器上并行地使用几个索引进行搜索。如果有多个物理磁盘驱动器和多个 CPU 内核，那么并发查询就能互不妨碍地执行。你可以假装有一些索引是远程的，然后配置 *searchd* 联系自身，从而在同一台机器上发起并行查询。

```
index distributed_sample
{
    type = distributed
    local = chunk1 # resides on HDD1
    agent = localhost:3312:chunk2 # resides on HDD2, searchd contacts itself
}
```

从客户端的视角来看，分布式索引跟本地索引完全没什么两样。这就允许你通过使用节点来代理其他的节点集的方式，来创建出一棵分布式索引的“树”。例如，第一级节点可以代理一定量的第二级节点的查询请求，结果就是，可以以任意的路径，本地搜索它们本身，或传递查询到其他节点。

聚合分片数据

构建一个可扩展的系统常常要涉及数据在不同物理 MySQL 服务器间的分片（分区）。这在第 11 章里已经深入讨论过了。

当数据以合适的粒度分片后，即使只是使用选择性的 WHERE（这应该非常快）获取几行数据的查询也意味着要关联到许多服务器，检查错误，以及在应用里将搜索结果合并在一起。Sphinx 减轻了这个问题的痛苦，因为所有必要的功能都在后台搜索进程里实现了。

考虑这样一个例子：有一个 1TB 大小的表，其中有 10 亿篇博客文章，通过用户 ID 分片到 10 个物理 MySQL 服务器上，这样给定用户的文章总是在同一台服务器上。只要查询是限制在单个用户上，一切都很好：我们根据用户 ID 先选定服务器，然后照常运作。 756

现在假定我们要实现一个归档分页功能来显示该用户的所有朋友发表的文章。我们怎么按发布日期排序显示“其他 sysbench 特性”的第 981 ~ 1000 个条目呢？大量朋友的数据很可能是在不同的服务器上。如果仅有 10 个朋友，那就有约 90% 的可能会用到 8 台以上服务器，如果是 20 个朋友，这个可能性就提高到了 99%。因此，对于大多数查询而言，我们需要关联到所有服务器。更糟糕的是，我们还要从每台服务器上拉 1000 篇文章，然后在应用程序里进行排序。按照本书前面所提供的建议，我们将数据裁减到只需要文章 ID 和时间戳。但是，仍然有 10 000 条记录要在应用程序里排序。许多现代脚本语言单在排序这一步骤上就会消耗掉大量的 CPU 时间。除此以外，我们或者要按顺序从每个服务器上获取结果记录（这会很慢），或者要编写一些代码构造并行查询线程（这很难实现和维护）。

在这样的情形下，采用 Sphinx 将比重新发明轮子显得更有意义。在本例中所要做的事情就是建立几个 Sphinx 实例，从每个表里映射出经常访问的文章属性——在这里就是文章 ID、用户 ID 和时间戳，然后在主 Sphinx 实例上查询第 981 ～ 1000 条记录，并按照发布日期排序，全部算起来大概是 3 行代码。这是更明智的扩展方法。

架构概要

Sphinx 是一个独立的程序集。两个主要程序如下。

indexer

　　这个程序用来从各种特定的资源上（例如 MySQL 的查询结果）获取文档，并据此创建全文索引。这是一个后台批处理任务，网站一般会定时运行它。

searchd

　　这是一个后台进程，用于查询 *indexer* 构建的索引。它为应用程序提供运行时支持。

Sphinx 的发布包里还包含有多种编程语言的 *searchd* 原生客户端 API（在本书写作时，这些语言包括 PHP、Python、Perl、Ruby 和 Java），以及在 MySQL 5.0 及以上版本中作为插件式存储引擎实现的客户端 SphinxSE。这些 API 和 SphinxSE 都可供客户端应用连接到 *searchd*，然后把查询语句传递过去，最终取回搜索结果。

每一个 Sphinx 全文索引都可以比作数据库里的一个表，与表里放置的一行行数据不同的是，Sphinx 索引包含的是文档。（Sphinx 也有一个单独的数据结构——多值属性，下文
757 会讲到。）每一个文档都有一个唯一的 32 位或 64 位整数标识符，取自数据表里的索引字段（例如，从主键列中取）。另外，每一个文档拥有一个或多个全文字段（每一个都对应于数据库里的一个文本字段）和数值属性。就像一个数据表一样，Sphinx 索引在所有文档里都有着一样的字段和属性。表 F-1 显示了数据表和 Sphinx 索引的相似之处。

表F-1：数据库结构和相应的Sphinx结构

数据库结构	Sphinx结构
`CREATE TABLE documents (`	`index documents`
`id int(11) NOT NULL auto_increment,`	`document ID`
`title varchar(255),`	`title field, full-text indexed`
`content text,`	`content field, full-text indexed`
`group_id int(11),`	`group_id attribute, sql_attr_uint`
`added datetime,`	`added attribute, sql_attr_timestamp`
`PRIMARY KEY (id)`	
`);`	

Sphinx 不存储数据库中的文本字体，只使用它们的内容来创建一个搜索索引。

安装综述

Sphinx 安装非常直观，一般包括如下步骤。

1. 从源码编译程序。

```
$ configure && make && make install
```

2. 创建一个配置文件：定义数据源和全文索引。
3. 初始索引化。
4. 启动 *searchd*。

在这之后，客户程序即拥有查询功能。

```
<?php
include ( 'sphinxapi.php' );
$cl = new SphinxClient ();
$res = $cl->Query ( 'test query', 'myindex' );
// use $res search result here
?>
```

唯一还没做的事情就是定时运行 *indexer* 来更新全文索引数据。在重建索引的时候，*searchd* 当前正在使用的索引还是全部可以使用的：*indexer* 会检测到索引正在使用，然后创建一个“影子”索引来代替。在索引创建完之后，它会通知 *searchd* 使用这个完成的副本。

全文索引存储在文件系统中（保存路径在配置文件里指定），存储为一种特定的“整体”形式，这种形式不适合做增量更新。通常的更新索引的方法就是全部重建。这个问题没 758
有看起来那么大，原因有以下几点。

- 创建索引的速度很快。在现在的硬件设备上，Sphinx 索引普通文本（不带 HTML 标记）的速度是 4 ～ 8MB/s。
- 可以把数据分割到几个索引里，下一小节里会讲到。每次运行 *indexer* 时只对需要更新的那部分数据进行索引重建。
- 无须对索引做“碎片整理”——它们本来就是为优化 I/O 而构建的，这能提高搜索速度。
- 数值属性可以直接更新，无须重建全部索引。

在未来的版本里，还会提供一个额外的索引后端，它将支持实时的索引更新。

典型的分区使用

下面详细讨论分区。最简单的分区模式是 *main+delta* 方法，对一个文档集创建两个索引。*main* 索引全部文档集，而 *delta* 只索引自上次 *main* 索引创建之后发生变更的文档。

这个模式与许多数据变更模式完全吻合。论坛、博客、电子邮件和新闻归档，以及垂直索引引擎都是很好的例子。那些存储库的大部分冷数据自创建后从不更新，只有很小一部分的热数据经常改变或增加。这意味着 delta 索引很小并且可以在需要时重建（例如，每隔 1 ~ 15 分钟一次）。这相当于只对新插入的行做索引。

你不需要重建索引来改变与文档关联的属性——可以通过 *searchd* 在线做。可以通过简单地在 main 索引上设置“deleted”特性来标记行已删除。因此，可以在 main 索引中对文档标记这个属性来处理更新，然后重建 delta 索引。对所有未标记为“deleted”的文档搜索会返回正确的结果集。

注意，索引文件可能来自任何 SELECT 语句的结果；不必来自单个 SQL 表。对 SELECT 语句没有限制。这意味着可以在数据库中建索引之前预处理结果。普通预处理例子包括：与其他表的联接，在线创建额外的字段，在索引时排除某些字段，以及生成一些值。

759 特别的功能特性

除“仅仅”索引和搜索数据库内容外，Sphinx 还提供几个其他的功能。下面是其中最重要的部分。

- 搜索和排位算法记录词的位置，并且查询阶段接近的文档内容，也会被计算在内。
- 可以把数值属性绑定到文档中，包括多值属性。
- 可以按属性值排序、过滤和分组。
- 可以创建与搜索查询关键字高亮的文档片段。
- 可以跨多台机器做分布式搜索。
- 可以优化查询，从相同的数据中产生几个结果集。
- 可以从 MySQL 中使用 SphinxSE 来访问搜索结果。
- 可以很好地调节 Sphinx 对服务器负载的影响。

我们在前面已经涉及了其中部分特性。本节将介绍剩下的几个特性。

近义词排位

Sphinx 能记住每一个文档内词语的位置，就像其他开源全文检索系统那样。但与它们不同的是，它使用位置对匹配度进行排序，返回更相关的结果。

有许多因素会影响到文档的最终排位。为了计算排位，其他许多系统只使用了关键字的频度：每一个关键字的出现次数。几乎被所有全文检索系统使用的经典的 BM25 权重函数[注1]就是把更多的权重分给这样一些词语，它们或者在特定的被检索文档里常常出现，或者很少在整个文档集里出现。BM25 返回的结果通常就是最终的排位值。

与之不同，Sphinx 也计算查询短语的近似度，就是文档内包含的查询子短语的最大长度，以词语为计数单位。举例来说，用短语“John Doe Jr”在带有文本“John Black, John White Jr, and Jane Dunne”的文档里搜索时，会产生一个 1 的短语近似度，因为按查询序列，没有两个词语一同出现在文档里。如果文档的文本是“Mr. John Doe Jr and friends”，那就是 3 的近似度，因为这三个查询词语依次出现在文档里。而文本“John Gray, Jane Doe Jr”会生成 2 的近似度，这归功于“Doe Jr”查询子短语。

在默认情况下，Sphinx 首先是用短语近似度排序的，其次才是经典的 BM25。这意味着 760
逐字的查询引用可以保证非常靠前，而由一个单独的词语引用刚好在它们下面，等等。

短语近似度是何时以及如何影响结果的呢？设想要在 1 000 000 页的文本里搜索短语“To be or not to be”。Sphinx 会将逐字引用的页面放在搜索结果的最前面，而其他基于 BM25 的系统会首先返回那些包含了最多的“to”、“be”、“or”和“not”的页面——那些包含了一个确切引用的页面会只因里面的“to”不够多，就被埋没在搜索结果的深处了。

当今许多主流的 Web 搜索引擎用关键字位置对结果进行排位也是一样的原理。在 Google 上搜索一个短语，它就会把最完美或接近完美包含匹配短语的文档放在结果的最上面，后面是“词袋”文档。

然而，分析关键词的位置会需要额外的 CPU 时间，有时可能会出于性能考虑而跳过这一步。在有些情况下，短语排位也会产生不受欢迎的、出乎意料的结果。例如，在云里搜索标签时没有关键字位置会更好一些：要查询的 tag 在文档里是否相邻也没什么区别。

为了顾及灵活性，Sphinx 提供了排位模式的选择。除了默认的近似度加 BM25 之外，还能选用多种其他类型的方法，包括只有 BM25 权值的、完全禁用权值的（如果不是使用排位做排序，它能提供很好的优化），等等。

支持属性

每一个文档都可以包含无限数目的数值属性。属性是用户指定的，能够根据特定任务的需要包含任何额外的信息。相应的例子包括：一篇博客文章的作者 ID、明细表里一个项目的价格、一个分类 ID，等等。

注 1： 详情参考 *http://en.wikipedia.org/wiki/Okapi_BM25*

属性依靠额外的过滤、排序和对搜索结果进行分组，以提高全文搜索的效率。在理论上，它们可以被存储在 MySQL 里，而在执行搜索时再取出来。但在实际应用中，如果全文检索要从 MySQL 里定位几百或几千行数据（这也不算多），检索它们将慢得不可接受。

Sphinx 支持两种存储属性的方式：内联到文档列表或者放在外部单独的文件里。内联要求所有属性值要在各索引里存储多次，每当有文档 ID 存入就会有一次。这会增加索引的大小，还会提升 I/O 数量，但也会减少 RAM 的使用。在外部对属性进行排序，需要在 *searchd* 启动时把它们预加载到 RAM 里。

属性通常都能被放入 RAM 中，因此，常见的做法就是把它们存储在外部。这可以使过滤、排序和分组更加快速，因为访问这些数据就相当于是一次快速的内存内查找。并且，只
761 有存放于外部的属性才能在运行时被更新。内联存储应该只在没有足够的空闲 RAM 来存储属性数据时使用。

Sphinx 也支持多值属性（MVA）。MVA 的内容由一个任意长的整数值列表组成，每个整数值对应一个文档。那些很好地利用了 MVA 的例子有标签 ID 列表、产品的分类和访问控制列表。

过滤

在全文搜索引擎里拥有对属性值的访问权可以让 Sphinx 在搜索时尽可能早地过滤和剔除候选匹配项。从技术上说，过滤检查发生在校验完文档是否包含了所有需要的关键字之后，但又在某个计算量很大的计算过程（例如排名）之前。因为有了这些优化，使用 Sphinx 把过滤和排序整合到全文搜索里要比在 Sphinx 中搜索而在 MySQL 中过滤结果快 10 ~ 100 倍。

Sphinx 支持两种类型的过滤，这与 SQL 里简单的 `WHERE` 条件很相似。

- 一个属性值匹配一个特定范围内的值（跟 `BETWEEN` 子句或数值比较相似）。
- 一个属性值匹配一个特定的值集合（跟 `IN()` 列表相似）。

如果过滤器有固定数量的值（用“集合”过滤器代替“范围”过滤器），并且这些值是可选择的，那么，用“伪关键字”替换掉整型值，并用全文内容而非属性的方式索引，这样是很有意义的。这同样适用于普通的数值属性和多值属性。在下文里我们会看到一些关于如何去做的例子。

Sphinx 能够使用过滤器来优化全扫描。它能记住在一小段连续的行块（默认是 128 行）中的最小和最大属性值，根据过滤条件很快地丢弃掉整个块。行是按照文档 ID 升序存储，

因此，这个优化工作最适合那些跟 ID 关联紧密的列。例如，如果有一个行插入时间戳，它会随着 ID 一起增长，那么，在这个时间戳上做带有过滤的全扫描会非常快。

SphinxSE 可插拔存储引擎

接收到来自 Sphinx 的全文搜索结果之后，几乎总会有一些涉及 MySQL 的额外工作要做——从最低限度来讲，Sphinx 索引里没有存储的文本列的值必须从 MySQL 里取得。因此，经常需要把 Sphinx 的搜索结果和其他 MySQL 表联接。

尽管可以把搜索结果里的文档 ID 写在一条查询语句中发送给 MySQL，但是，这种方法会导致既不太简洁也不太高效的代码。对于量非常大的场景，应该考虑使用 SphinxSE，这是一个可编译到 MySQL 5.0 或更新的版本里的可插拔存储引擎，也可作为一个插件加载到 MySQL 5.1 或更新的版本里。 762

SphinxSE 可以让程序员从 MySQL 里面查询 *searchd* 和访问搜索结果。用法非常简单，只要在创建表时加上 ENGINE=SPHINX 子句（还有一个可选的 CONNECTION 子句，用于 Sphinx 服务器不在默认路径时重新定位服务器），然后就可以在表上运行查询了。

```
mysql> CREATE TABLE search_table (
    ->     id         INTEGER NOT NULL,
    ->     weight     INTEGER NOT NULL,
    ->     query      VARCHAR(3072) NOT NULL,
    ->     group_id   INTEGER,
    ->     INDEX(query)
    -> ) ENGINE=SPHINX CONNECTION="sphinx://localhost:3312/test";
Query OK, 0 rows affected (0.12 sec)

mysql> SELECT * FROM search_table WHERE query='test;mode=all' \G
*************************** 1. row ***************************
      id: 123
  weight: 1
   query: test;mode=all
group_id: 45
1 row in set (0.00 sec)
```

每个 SELECT 以 WHERE 子句中的 query 列的方式传递给 Sphinx 查询。Sphinx *searchd* 服务器返回查询结果，然后 SphinxSE 存储引擎会把它们翻译成 MySQL 的结果返回给该 SELECT 语句。

其中的查询可能会包含 JOIN，把别的存储引擎上的其他表联接进来。

SphinxSE 支持大部分原本在 API 里才可用的搜索选项。你可以通过插入额外子句到查询字符串的方式设定类似过滤和范围限制选项。

```
mysql> SELECT * FROM search_table WHERE query='test;mode=all;
    -> filter=group_id,5,7,11;maxmatches=3000';
```

通过 API 返回的关于每一个查询和每一个词语的统计信息也可以用 SHOW STATUS 访问。

```
mysql> SHOW ENGINE SPHINX STATUS \G
*************************** 1. row ***************************
  Type: SPH INX
  Name: stats
Status: total: 3, total found: 3, time: 8, words: 1
*************************** 2. row ***************************
  Type: SPHINX
  Name: words
Status: test:3:5
2 rows in set (0.00 sec)
```

763 甚至在使用 SphinxSE 的时候，经验法则仍然允许 *searchd* 执行排序、过滤和分组——也就是说，把所有需要的子句加到查询字符串里，而不是使用 WHERE、ORDER BY 和 GROUP BY。这对于 WHERE 条件来说尤为重要，其原因是 SphinxSE 仅仅是 *searchd* 的一个客户端，而不是一个全能的内嵌搜索库。因此，你还是要把所有东西都传递给 Sphinx 引擎以获得最好的性能表现。

高级性能控制

索引和搜索操作都会显著地加重搜索服务器或数据库服务器的负担。幸运的是，有很多设置可用以限制来自 Sphinx 的负载。

indexer 的查询会引发不期望的数据库端负载，它们或者是因为使用到的锁彻底地减慢了 MySQL 的运转速度，或者只是因为出现得太快而与其他并发查询竞争资源。

第一个例子就是 MyISAM 的一个声名狼藉的问题，当长时间的读锁定表时，会影响到其他后续的读和写——你不能简单地在生产服务器上执行 SELECT * FROM big_table，因为会有干扰其他所有操作的风险。为了绕开这个问题，Sphinx 提供了区间查询。与配置单个巨大查询不同，你可以指定一个可以快速计算可索引的行区间的查询，以及另外一个以很小的块逐批往外拉数据的查询。

```
sql_query_range  = SELECT MIN(id),MAX(id) FROM documents
sql_range_step   = 1000
sql_query        = SELECT id, title, body FROM documents \
                   WHERE id>=$start AND id<=$end
```

这个特性在对 MyISAM 表索引的时候非常有用，但也应该考虑到使用 InnoDB 表的情况。虽然 InnoDB 在运行一个大数据量 SELECT 的时候不会锁定表，也不会延误其他查询的执行，但是，由于它的 MVCC 架构，它还是会使用到很多的机器资源。1 000 个多版本的事务，

每个事务会涉及 1 000 行数据，总开销也小于单独一个要长时间运行的涉及 100 万行数据的事务。

第二种加重负载的可能发生在 *indexer* 能够比 MySQL 更快地处理数据时。在这样的情形下，也应该使用范围查询。`sql_ranged_throttle` 选项会强制 *indexer* 在后续的查询步骤之间休眠给定的一段时间（以毫秒计算），这虽然会增加索引建立的时间，但也会减轻 MySQL 的负担。

如果有足够兴趣，你还可以看一个特别的案例，配置 Sphinx 以达到相反效果：为了缩短索引建立时间，就将更多的负载加到 MySQL 上面。当 *indexer* 和数据库之间的连接是 100MB，行都被很好地压缩时（典型的文本数据），MySQL 的压缩协议能缩短整体的索引时间。随之而来的是另外一个开销增加了：为了压缩和解压缩网络上传输的行数据，
MySQL 端和 *indexer* 端各自都会使用到更多的 CPU 时间。但是，整个索引时间因为减 764
少了网络数据流量而能够缩短 20% ~ 30%。

集群上的搜索偶尔也会遇到过载问题，因此，Sphinx 提供了一些方法以避免 *searchd* 跑飞。

首先，`max_children` 选项可以简单地限制一下能够并发运行的查询数量，当达到极限时告诉客户端重试。

其次，还有查询级别的限制。可以设定查询运行的时候，或者是在找到预定个数的匹配项时就停止，或者是用完指定长度时间之后就停止。这两个条件分别通过调用 `SetLimits()` 和 `SetMaxQueryTime()` API 来实现。这是针对每一个查询设置的，所以，可以确保更重要的查询总是能够彻底完成。

最后，定时运行 *indexer* 会引起额外 I/O 的突增，随之会影响到 *searchd* 间歇性地速度减慢。为了防止这种情况发生，Sphinx 里有相应的选项来限制 *indexer* 的磁盘 I/O。`max_iops` 强加了两次 I/O 操作之间的最小延迟时间，以确保每一秒里不会有超过 `max_iops` 的磁盘操作会被执行。但是，有时一个单独的操作也会很占时间，比如有一个 100MB 数据量的 `read()` 调用。`max_iosize` 选项会处理这种情况，它可以保证每一次磁盘读或者写的长度都被限制在指定的范围之内。更大的操作会被自动地分解成小型操作，然后，这些小型的操作由 `max_iops` 设置控制。

实际应用案例

每个描述的特性都可以在产品部署中成功找到。下面的小节回顾了几个现实的 Sphinx 部署，简要描述了网站的情况和一些实现细节。

Mininova.org 上的全文搜索

Mininova（*http://www.mininova.org*）是一个流行的 BT 种子搜索引擎，它清楚地展示了如何“仅仅”优化全文检索。Sphinx 替代了几个基于 MySQL 主从复制的备库内建的全文索引的 MySQL，因为它们不能很好地处理负载。替换之后，搜索服务器负载很轻；当前平均负载在 0.3~04。

数据库规模和负载量如下。

- 网站的数据库很小，大概 30 万 ~50 万条记录，300MB~500MB 索引量。
- 网站的负载非常高：在写作本书的时候每天大约 800 万 ~1000 万次查询。

数据绝大部分是由用户提供的文件名，常常没有合适的标记符号。因为这个原因，采用
765 了前缀索引而不是整词索引。结果索引比不这样做要大好几倍，但仍然足够小，可以快速地构建，并且数据可以高效地缓存。

对 1 000 个最频繁的查询的搜索结果在应用端缓存起来。总查询中有大概 20%~30% 由缓存提供服务。由于“长尾”查询分布，缓存再大一些并不会起太多作用。

为了高可用，网站使用两个服务器和一个完整的全文索引复制服务器。索引每隔几分钟从头重建。建索引耗时小于一分钟，因此构建更复杂的模式没有意义。

下面是从这个案例中学到的：

- 在应用中缓存结果非常有帮助。
- 巨大的缓存可能没有必要，甚至对繁忙的应用也是如此。只需要 1000~10000 个条目就足够了。
- 对于近 1GB 大小的数据库，简单周期性地重建索引而不是创建更复杂的模式是没有问题的，即使对于繁忙的网站也是如此。

BoardReader.com 上的全文搜索

Mininova 是个负载格外高的项目案例——数据量不是太多，但有许多对数据的查询。BoardReader（*http://www.boardreader.com*）恰好相反：一个论坛搜索引擎，在非常大的数据集中执行非常少量的查询。Sphinx 替代了商业的全文搜索引擎，对 1GB 的数据集合每次查询将需 10s。Sphinx 可使 BoardReader 在数据量和查询的吞吐量上很好地扩展。

下面是一些常规信息。

- 在数据库中有 10 亿个文档和 1.5TB 的文本。
- 有大概 50 万个页面视图，每天的查询量在 70 万 ~100 万。

在写作本书的时候，搜索集群由 6 台服务器组成，每台有 4 个逻辑 CPU（两个 Xeon 双核），有 16GB 的 RAM 和 0.5TB 的磁盘空间。数据库本身存储在一个分开的集群上。搜索集群只用来做索引和搜索。

6 台服务器中每台有 4 个 *searchd* 实例，因此所有 4 个核都被使用。4 个实例其中之一聚集了其他三个上的结果。总共 24 个 *searchd* 实例。数据在它们中间平均分布。每个 *searchd* 副本携带几个索引，大概超过总数据的 1/24（大约 60GB）。

来自 6 个“第一层” *searchd* 节点的搜索结果由另外一个运行在 Web 服务器前端的 *searchd* 实例所聚集。这个实例只携带几个纯分布的索引，指向 6 个搜索集群服务器，但本地并没有数据。

为什么每个节点上要有 4 个 *searchd* 实例？为什么不是每个服务器上只一个 *searchd* 实 766
例，配置它运载 4 个索引块，自己和自己通信，就像远程服务器那样来利用多核 CPU，一如我们之前建议的那样？有四个实例而不是一个有它的好处。首先，它减少了启动时间。有几个 GB 的属性数据需要预先加载到 RAM 中；在同一时间启动几个后台进程可以并行执行。其次，这可增加可用性。在 *searchd* 失败或更新的情形下，整个索引中只有 1/24 不可访问，而不是 1/6。

在搜索集群的 24 个实例里，我们使用基于时间的分片来进一步减少负载量。许多查询只需运行在最近的数据之上，因此，数据可以被分成 3 个不相交的索引集：最近一个星期的数据、最近 3 个月的数据和全部数据。这些索引分布在每个实例所在服务器的几块物理磁盘上。通过这个方法，每个实例都拥有了自己的 CPU 和物理磁盘驱动器，且互不干扰。

本地的 *cron* 任务会定时更新索引，跨网络从 MySQL 上拉数据，但是只在本地创建索引文件。

使用几块明确独立的“裸”磁盘被证明快于单独的 RAID 卷。裸磁盘能够控制哪一些文件存储到哪块物理磁盘上，而在 RAID 里却不一样，控制器将决定哪一个数据块存储在哪一块物理磁盘上。裸磁盘也能保证在不同的索引块上充分使用并行 I/O，但基于 RAID 的并发查询仍然受制于 I/O 层级。我们在此处选择的是没有冗余的 RAID 0，这是因为我们不关心磁盘故障；我们可以在搜索节点很方便地重建索引。当需要提高可靠性时，也可以使用几个 RAID 1（镜像）卷提供跟裸盘相同的吞吐量。

从 BoardReader 那里了解到的另一个有趣的事情是 Sphinx 版本升级是如何执行的。显然，整个集群是不可能停掉的，因此，向后兼容非常重要。幸运的是，Sphinx 提供了这个功能——新版本的 *searchd* 一般都能读出旧版本的索引文件，也总能跨网络跟旧的客户端

通信。要注意的是第一层上用来聚集搜索结果的节点对于第二层节点而言就像客户端，而由第二层节点执行实际的大多数搜索。所以，第二层上的节点首先升级，然后是第一层上的节点，最后才是 Web 前端。

在本例中学到的经验如下。

- 对于超大型数据库的格言是：分片，分片，分片，并行。
- 在大规模的搜索应用里，组织 *searchd* 为多层树状结构。
- 尽可能地针对全部数据的一小部分建立优化的索引。
- 明确地将文件映射到磁盘而不是依靠 RAID 控制器。

767 Sahibinden.com 对 SELECT 的优化

Sahibinden（*http://www.sahibinden.com*）是土耳其一家领先的在线拍卖网站，存在着大量的性能问题，包括全文搜索性能。在部署了 Sphinx 并对查询做了一些分析之后，我们发现 Sphinx 高频执行应用相关的过滤查询，要比 MySQL 快许多——即使在 MySQL 中有对相关列的索引。另外，运用 Sphinx 于非全文搜索时，可产生易于编写和支持的统一的应用代码。

MySQL 的表现不佳是因为每个单独列的选择性不足以明显地缩小搜索空间。事实上，要创建和维护所有需要的索引几乎不可能，因为有太多的列需要它们。产品信息表大约有 100 个列，从技术上讲，Web 应用可能使用任一列来过滤或排序。

在这个“热门”的产品表上的插入和更新都是龟速，因为有太多的索引需要随之更新。

基于这些原因，要应付产品信息表上的所有 `SELECT` 查询，而不仅仅是全文搜索查询，Sphinx 就是一个自然的选择。

网站数据库的大小和负载量如下。

- 数据库里包含了大约 400 000 行记录，500MB 数据。
- 负载量大约是每天 300 万个查询。

为了模拟普通的带 `WHERE` 条件的 `SELECT` 查询，Sphinx 在建索引过程中把一些特别的关键字写在全文索引里。这些关键字都形如 `__CATN___`，这里的 *N* 可以用相应的分类 ID 来代替。这个替代发生在 MySQL 查询中运行带 `CONCAT()` 函数的索引之时，因此源数据不会被修改。

索引需要尽可能频繁地重建。我们是固定每分钟重建一次。在多 CPU 环境里每次重建的时间是 9 ~ 15s，因此，早先讨论过的 *main+delta* 方案是不需要的。

实践证明，PHP API 在解析带有大量属性的结果集时，会花费相当数量的时间（每个查询大约 7 ~ 9ms）。通常，这个开销不会构成问题，因为全文搜索的开销，特别是在大数据集上执行时，会高于这个解析。为了能减少这个因素的影响，索引被分隔成一对对的："轻量"的有 34 个常用的属性，而"完整"的有全部 99 个属性。

其他可能的解决办法是使用 SphinxSE 或者实现一个能够把特定的列读到 Sphinx 里的功能。然而，使用两个索引的方法是目前实现起来最快的，时间也是重要因素。

以下是我们从这个案例里学到的经验。 768

- 有时，Sphinx 上的全扫描的执行效率要好过 MySQL 的索引读取。
- 对于选择性条件,用"伪关键字"代替属性过滤之后,全文搜索引擎能做更多的工作。
- 脚本语言里的 API 在某些极端但现实的条件下可能会成为性能瓶颈。

BoardReader.com 对 GROUP BY 的优化

对 BoardReader 服务的改进需要统计超链接数，并且要根据关联数据创建不同的报表。例如，报表之一就是要显示出最近一个星期来链接数排在最前面的 *N* 个二级域名。另外一个统计链接到指定站点例如 YouTube 的最前面 *N* 个二级和三级域名。用来创建这些报表的查询具有下列这些常见特征。

- 它们总是按域来分组的。
- 它们按每个组里的链接数排序，或者是以每个组里的唯一域名的链接数。
- 它们要处理大量的数据（接近几百万行记录），但是，最后生成的带有最佳分组的结果集往往又很小。
- 近似的结果也能接受。

在原型测试环节里,MySQL 大概花了 300s 来执行这些查询。理论上,通过数据分片技术,把它们分拆到各个服务器执行，再在应用里用人工方式聚合查询结果，还可能将这些查询的时间优化到 10s 左右。但是，这需要构建一个复杂的架构，即使分片的实现也远不是那么直接明了。

因为已经成功地使用 Sphinx 对搜索负载进行过分布式处理，所以，我们决定还使用 Sphinx 来实现一个近似的分布式的 GROUP BY。这就要求在建立索引之前预处理这些数据,把所有感兴趣的子串转换为单独的"词语"。以下就是一个示例 URL 在预处理前后的样子。

```
source_url      = http://my.blogger.com/my/best-post.php
processed_url   = my$blogger$com, blogger$com, my$blogger$com$my,
                  my$blogger$com$my$best, my$blogger$com$my$best$post.php
```

美元符号（$）仅仅是对 URL 分隔符统一的替换，这样搜索就能工作在任何 URL 部分，

域或者路径。这种预处理能析取所有“感兴趣”的子串成为单独的关键字，这样搜索起来是最快的。从技术上说，我们可以采用短语查询或者前缀索引，但那些都会导致更大的索引，速度也更慢。

链接是在构建索引时预处理的，使用了特定的 MySQL UDF。在这个任务里，我们还定制了一个计算唯一性值的功能用来加速 Sphinx。在此之后，我们就能把查询全部移到搜索集群里，简单地分发，同时明显减少查询延迟。

769 数据库的大小和负载量如下。

- 里面约有 1.5 亿 ~2 亿行记录，预处理之后会生成 50~100GB 数据。
- 负载是每天大概 60 000~100 000 个 GROUP BY 查询。

用于分布的 GROUP BY 的索引也是部署在先前我们提到的同一个搜索集群上——有着 6 台机器，24 个逻辑 CPU。相对存储了 1.5TB 文本的数据库而言，这只是主要搜索负载的一个零头。

Sphinx 替代了 MySQL 的精确、缓慢、单 CPU 的计算方式，转而用近似、快速、分布式的计算方式。所有会引入近似错误的因素都会存在：输入数据常常包含太多的行，以至于无法放入“排序缓冲区”里（我们使用的是一个固定的 10 万行的 RAM），我们使用了 COUNT(DISTINCT)，结果集是通过网络聚合的。除此以外，对于结果集里的前 10 ~ 1000 组——这常常是报表实际所需的——能够有 99% ~ 100% 的准确率。

索引的数据跟普通全文搜索用的数据很不一样。它里面有巨额的文档和关键字，尽管文档都非常小。文档的编号也是非连续的，因为它使用的是一种特殊的编号约定（综合了源服务器、源表和主键），因而无法用 32 位来存储。巨大数量的搜索“关键字”也会引发频繁的 CRC32 冲突（Sphinx 用 CRC32 把关键字映射到内部词语的 ID）。因为这些原因，我们只能被迫在内部到处使用 64 位的标识符。

系统目前的性能很让人满意。对于最复杂的域名，完成一次查询的时间通常是 0.1 ~ 1.0s。

以下就是从这个案例里学到的经验。

- 对于 GROUP BY 查询，可以牺牲掉一些精度以获取更快的速度。
- 对于大量文本的集合或者适当大小的特殊数据集，可能要用 64 位标识符。

Grouply.com 对联接查询的优化

Grouply（*http://www.grouply.com*）构建了一个基于 Sphinx 的解决方案来搜索几百万行的标签信息数据库，其中利用了 Sphinx 的 MVA 支持。为了高可扩展性，数据库被分解

到许多物理服务器上，因而对跨不同服务器的表的查询是必需的。然而任意大规模的联接是不可能的，因为会有太多的服务器、数据库和表参与执行。

Grouply 使用 Sphinx 的 MVA 特性来存储消息标签。标签列表通过 PHP 的 API 从 Sphinx 集群中获取。这替代了从 MySQL 服务器来的多个顺序的 SELECT。同时，为了减少 SQL 查询的数量，某些特定用于展现的数据（例如，最后读取消息的一小部分用户的列表）同样存储在单独的 MVA 属性中，并且通过 Sphinx 访问。 770

这里有两项创新点：使用了 Sphinx 预先构建 JOIN 结果，并且使用分布式功能合并了分散在各个数据块上的数据。只使用 MySQL 是不可能做到这些的。高效的归并要求数据能够分拆到尽可能少的物理服务器和表上，但这又会损害可扩展性和可扩充性。

从本案例中学到的经验如下。

- Sphinx 能够用于有效地聚合高度分区化的数据。
- MVA 能够用于存储和优化预先构建的 JOIN 结果。

总结

在本附录里，我们只是简要地讨论了 Sphinx 全文搜索系统。为了缩短篇幅，我们特意略过了关于 Sphinx 其他许多功能特性的讨论，例如对 HTML 索引的支持、对 MyISAM 有更好支持的范围搜索、词法和同义词的支持、前缀和中缀索引以及 CJK 索引。不过，这个附录应该已经给了你一些关于 Sphinx 如何高效解决真实世界各种问题的启发。它并不限于做全文搜索，还能解决许多传统 SQL 的老难题。

Sphinx 既不是一颗银弹，也不是 MySQL 的一个替代品，但是在许多应用案例里（对于现代 Web 应用已经变得很常见），它可以被当作 MySQL 很有用的补充。你可以简单地用它来减轻一些工作的负担，甚至为你的应用创造新的可能性。

你可以从 *http://www.sphinxsearch.com* 下载 Sphinx，另外，别忘记分享你自己的使用心得。

索引†

Symbols

32-bit architecture, 390
404 errors, 614, 617
451 Group, 549
64-bit architecture, 390
:= assign operator, 249
@ user variable, 253
@@ system variable, 334

A

ab tool, Apache, 51
Aborted_clients variable, 688
Aborted_connects variable, 688
access time, 398
access types, 205, 727
ACID transactions, 6, 551
active caches, 611
active data, keeping separate, 554
active-active access, 574
Adaptec controllers, 405
adaptive hash indexes, 154, 703
Adaptive Query Localization, 550, 577
Address Resolution Protocol (ARP), 560, 584
Adminer, 666
admission control features, 373
advanced performance control, 763
after-action reviews, 571
aggregating sharded data, 755
Ajax, 607
Aker, Brian, 296, 679
Akiban, 549, 552
algebraic equivalence rules, 217
algorithms, load-balancing, 562
ALL_O_DIRECT variable, 363
ALTER TABLE command, 11, 28, 141–144, 266, 472, 538
Amazon EBS (Elastic Block Store), 589, 595
Amazon EC2 (Elastic Compute Cloud), 589, 595–598
Amazon RDS (Relational Database Service), 589, 600
Amazon Web Services (AWS), 589
Amdahl scaling, 525
Amdahl's Law, 74, 525
ANALYZE TABLE command, 195
ANSI SQL isolation levels, 8
Apache ab, 51
application-level optimization
  alternatives to MySQL, 619
  caching, 611–618
  common problems, 605–607
  extending MySQL, 618
  finding the optimal concurrency, 609
  web server issues, 608
approximations, 243
Archive storage engine, 19, 220
Aria storage engine, 23, 681
ARP (Address Resolution Protocol), 560, 584
Aslett, Matt, 549
Aspersa (see Percona Toolkit)
asynchronous I/O, 702
asynchronous replication, 447
async_unbuffered, 364
atomicity, 6
attributes, 749, 760
audit plugins, 297
auditing, 622
authentication plugins, 298
auto-increment keys, 578

† ：索引所列页码为本书英文版页码，请参照用“▭”表示的原书页码。

AUTOCOMMIT mode, 10
autogenerated schemas, 131
AUTO_INCREMENT, 142, 275, 505, 545
availability zone, 572
AVG() function, 139
AWS (Amazon Web Services), 589

B

B-Tree indexes, 148, 171, 197, 217, 269
Background Patrol Read, 418
Backup & Recovery (Preston), 621
backup load, 626
backup time, 626
backup tools
  Enterprise Backup, MySQL, 658
  mydumper, 659
  mylvmbackup, 659
  mysqldump, 660
  Percona XtraBackup, 658
  Zmanda Recovery Manager, 659
backups, 425, 621
  binary logs, 634–636
  data, 637–648
  designing a MySQL solution, 624–634
  online or offline, 625
  reasons for, 622
  and replication, 449
  scripting, 661–663
  snapshots not, 646
  and storage engines, 24
  tools for, 658–661
balanced trees, 223
Barth, Wolfgang, 668
batteries in SSDs, 405
BBU (battery backup unit), 422
BEFORE INSERT trigger, 287
Beginning Database Design (Churcher), 115
Bell, Charles, 519
Benchmark Suite, 52, 55
BENCHMARK() function, 53
benchmarks, 35–37
  analyzing results, 47
  capturing system performance and status, 44
  common mistakes, 40, 340
  design and planning, 41
  examples, 54–66
  file copy, 718
  flash memory, 403
  getting accurate results, 45
  good uses for, 340
  how long to run, 42
  iterative optimization by, 338
  MySQL versions read-only, 31
  plotting, 49
  SAN, 423
  strategies, 37–40
  tactics, 40–50
  tools, 51–53
  what to measure, 38
BerkeleyDB, 30
BIGINT type, 117
binary logs
  backing up, 634–636
  format, 635
  master record changes (events), 449, 496
  purging old logs safely, 636
  status, 688
binlog dump command, 450, 474
binlog_do_db variable, 466
binlog_ignore_db variable, 466
Birthday Paradox, 156
BIT type, 127
bit-packed data types, 127
bitwise operations, 128
Blackhole storage engine, 20, 475, 480
blktrace, 442
BLOB type, 21, 121, 375
blog, MySQL Performance, 23
BoardReader.com, 765, 768
Boolean full-text searches, 308
Boost library, 682
Bouman, Roland, 281, 287, 667
buffer pool
  InnoDB, 704
  size of, 344
buffer threads, 702
built-in MySQL engines, 19–21
bulletin boards, 27
burstable capacity, 42
bzip2, 716

C

cache hits, 53, 316, 340, 395
CACHE INDEX command, 351
cache tables, 136
cache units, 396
cachegrind, 78

caches
  allocating memory for, 349
  control policies, 614
  hierarchy, 393, 616
  invalidations, 322
  misses, 321, 352, 397
  RAID, 419
  read-ahead data, 421
  tuning by ratio, 340
  writes, 421
Cacti, 430, 669
Calpont InfiniDB, 23
capacitors in SSDs, 405
capacity planning, 425, 482
cardinality, 160, 215
case studies
  building a queue table, 256
  computing the distance between points, 258
  diagnostics, 102–110
  indexing, 189–194
  using user-defined functions, 262
CD-ROM applications, 27
Change Data Capture (CDC) utilities, 138
CHANGE MASTER TO command, 453, 457, 489, 491, 501
CHAR type, 120
character sets, 298, 301–305, 330
character_set_database, 300
CHARSET() function, 300
CHAR_LENGTH() function, 304
CHECK OPTION variable, 278
CHECK TABLES command, 371
CHECKSUM TABLE command, 488
chunk size, 419
Churcher, Clare, 115
Circonus, 671
circular replication, 473
Cisco server, 598
client, returning results to, 228
client-side emulated prepared statements, 295
client/server communication settings, 299
cloud, MySQL in the, 589–602
  benchmarks, 598
  benefits and drawbacks, 590–592
  DBaaS, 600
  economics, 592
  four fundamental resources, 594
  performance, 595–598
  scaling and HA, 591
Cluster Control, SeveralNines, 577
Cluster, MySQL, 576
clustered indexes, 17, 168–176, 397, 657
clustering, scaling by, 548
Clustrix, 549, 565
COALESCE() function, 254
code
  backing up, 630
  stored, 282–284, 289
Codership Oy, 577
COERCIBILITY() function, 300
cold or warm copy, 456
Cole, Jeremy, 85
collate clauses, 300
COLLATION() function, 300
collations, 119, 298, 301–305
collisions, hash, 156
column-oriented storage engines, 22
command counters, 689
command-line monitoring with innotop, 672–676
command-line utilities, 666
comments
  stripping before compare, 316
  version-specific, 289
commercial monitoring systems, 670
common_schema, 187, 667
community storage engines, 23
complete result sets, 315
COMPRESS() function, 377
compressed files, 715
compressed MyISAM tables, 19
computations
  distance between points, 258–262
  integer, 117
  temporal, 125
Com_admin_commands variable, 689
CONCAT() function, 750, 767
concurrency
  control, 3–6
  inserts, 18
  measuring, 39
  multiversion concurrency control (MVCC), 12
  need for high, 596
configuration
  by cache hit ratio, 340
  completing basic, 378–380

creating configuration files, 342–347
InnoDB flushing algorithm, 412
memory usage, 347–356
MySQL concurrency, 371–374
workload-based, 375–377
connection management, 2
connection pooling, 561, 607
connection refused error, 429
connection statistics, 688
CONNECTION_ID() function, 257, 289, 316, 502, 635
consistency, 7
consolidation
scaling by, 547
storage, 407, 425
constant expressions, 217
Continuent Tungsten Replicator, 481, 516
CONVERT() function, 300
Cook, Richard, 571
correlated subqueries, 229–233
corrupt system structures, 657
corruption, finding and repairing, 194, 495–498
COUNT() function optimizations, 206, 217, 241–243, 292
counter tables, 139
counters, 686, 689
covering indexes, 177–182, 218
CPU-bound machines, 442
CPUs, 56, 70, 388–393, 594, 598
crash recovery, 25
crash testing, 422
CRC32() function, 156, 541
CREATE and SELECT conversions, 28
CREATE INDEX command, 353
CREATE TABLE command, 184, 266, 353, 476, 481
CREATE TEMPORARY TABLE command, 689
cron jobs, 288, 585, 630
crontab, 504
cross-data center replication, 475
cross-shard queries, 535, 538
CSV format, 638
CSV logging table, 601
CSV storage engine, 20
CURRENT_DATE() function, 316
CURRENT_USER() function, 316, 460
cursors, 290
custom benchmark suite, 339
custom replication solutions, 477–482

D

daemon plugins, 297
dangling pointer records, 553
data
archiving, 478, 509
backing up nonobvious, 629
changes on the replica, 500
consistency, 632
deduplication, 631
dictionary, 356
distribution, 448
fragmentation, 197
loss, avoiding, 553
optimizing access to, 202–207
scanning, 269
sharding, 533–547, 565, 755
types, 115
volume of and search engine choice, 27
Data Definition Language (DDL), 11
Data Recovery Toolkit, 195
data types
BIGINT, 117
BIT, 127
BLOB, 21, 121, 375
CHAR, 120
DATETIME, 117, 126
DECIMAL, 118
DOUBLE, 118
ENUM, 123, 130, 132, 282
FLOAT, 118
GEOMETRY, 157
INT, 117
LONGBLOB, 122
LONGTEXT, 122
MEDIUMBLOB, 122
MEDIUMINT, 117
MEDIUMTEXT, 122
RANGE COLUMNS, 268
SET, 128, 130
SMALLBLOB, 122
SMALLINT, 117
SMALLTEXT, 122
TEXT, 21, 121, 375
TIMESTAMP, 117, 126, 631
TINYBLOB, 122
TINYINT, 117

TINYTEXT, 122
VARCHAR, 119, 124, 131, 513
data=journal option, 433
data=ordered option, 433
data=writeback option, 433
Database as a Service (DBaaS), 589, 600
database servers, 393
Database Test Suite, 52
Date, C. J., 255
DATETIME type, 117, 126
DBaaS (Database as a Service), 589, 600
dbShards, 547, 549
dbt2 tool, 52, 61
DDL (Data Definition Language), 11
deadlocks, 9
Debian, 683
debug symbols, 99
debugging locks, 735–744
DECIMAL type, 118
deduplication, data, 631
"degraded" mode, 485
DELAYED hint, 239
delayed key writes, 19
delayed replication, 654
DELETE command, 267, 278
delimited file backups, 638, 651
DeNA, 618
denormalization, 133–136
dependencies on nonreplicated data, 501
derived tables, 238, 277, 725
DETERMINISTIC variable, 284
diagnostics, 92
capturing diagnostic data, 97–102
case study, 102–110
single-query versus server-wide problems, 93–96
differential backups, 630
directio() function, 362
directory servers, 542
dirty reads, 8
DISABLE KEYS command, 143, 313
disaster recovery, 622
disk queue scheduler, 434
disk space, 511
disruptive innovations, 31
DISTINCT queries, 135, 219, 244
distributed (XA) transactions, 313
distributed indexes, 754
distributed memory caches, 613
distributed replicated block device (DRBD), 494, 568, 574, 581
distribution master and replicas, 474
DNS (Domain Name System), 556, 559, 572, 584
document pointers, 306
Domain Name System (DNS), 556, 559, 572, 584
DorsalSource, 683
DOUBLE type, 118
doublewrite buffer, 368, 412
downtime, causes of, 568
DRBD (distributed replicated block device), 494, 568, 574, 581
drinking from the fire hose, 211
Drizzle, 298, 682
DROP DATABASE command, 624
DROP TABLE command, 28, 366, 573, 652
DTrace, 431
dump and import conversions, 28
duplicate indexes, 185–187
durability, 7
DVD-ROM applications, 27
dynamic allocation, 541–543
dynamic optimizations, 216
dynamic SQL, 293, 335–337

E

early termination, 218
EBS (Elastic Block Store), Amazon, 589, 595
EC2 (Elastic Compute Cloud), 589, 595–598
edge side (ESI), 608
Elastic Block Store (EBS), Amazon, 589, 595
Elastic Compute Cloud (EC2), Amazon, 589, 595–598
embedded escape sequences, 301
eMLC (enterprise MLC), 402
ENABLE KEYS command, 313
encryption overhead, avoiding, 716
end_log_pos, 635
Enterprise Backup, MySQL, 457, 624, 627, 631, 658
enterprise MLC (eMLC), 402
Enterprise Monitor, MySQL, 80, 670
ENUM type, 123, 130, 132, 282
equality propagation, 219, 234
errors
404 error, 614, 617
from data corruption or loss, 495–498

ERROR 1005, 129
ERROR 1168, 275
ERROR 1267, 300
escape sequences, 301
evaluation order, 253
Even Faster Websites (Souders), 608
events, 282, 288
exclusive locks, 4
exec_time, 636
EXISTS operator, 230, 232
expire_logs_days variable, 381, 464, 624, 636
EXPLAIN command, 89, 165, 182, 222, 272, 277, 719–733
explicit allocation, 543
explicit invalidation, 614
explicit locking, 11
external XA transactions, 315
extra column, 732

F

Facebook, 77, 408, 592
fadvise() function, 626
failback, 582
failover, 449, 582, 585
failures, mean time between, 570
Falcon storage engine, 22
fallback, 582
fast warmup feature, 351
FathomDB, 602
FCP (Fibre Channel Protocol), 422
fdatasync() function, 362
Federated storage engine, 20
Fedora, 683
fencing, 584
fetching mistakes, 203
Fibre Channel Protocol (FCP), 422
FIELD() function, 124, 128
FILE () function, 600
FILE I/O, 702
files
  consistency of, 633
  copying, 715
  descriptors, 690
  transferring large, 715–718
filesort, 226, 377
filesystems, 432–434, 573, 640–648
filtered column, 732
filtering, 190, 466, 564, 750, 761
fincore tool, 353
FIND_IN_SET() function, *128*
fire hose, drinking from the, 211
FIRST() function, 255
first-write penalty, 595
Five Whys, 571
fixed allocation, 541–543
flapping, 583
flash storage, 400–414
Flashcache, 408–410
Flexviews tools, 138, 280
FLOAT type, 118
FLOOR() function, 260
FLUSH LOGS command, 492, 630
FLUSH QUERY CACHE command, 325
FLUSH TABLES WITH READ LOCK command, 355, 370, 490, 494, 626, 644
flushing algorithm, InnoDB, 412
flushing binary logs, 663
flushing log buffer, 360, 703
flushing tables, 663
FOR UPDATE hint, 240
FORCE INDEX hint, 240
foreign keys, 129, 281, 329
Forge, MySQL, 667, 710
FOUND_ROWS() function, 240
fractal trees, 22, 158
fragmentation, 197, 320, 322, 324
free space fragmentation, 198
FreeBSD, 431, 640
“freezes”, 69
frequency scaling, 392
.frm file, 14, 142, 354, 711
FROM_UNIXTIME() function, 126
fsync() function, 314, 362, 368, 656, 693
full-stack benchmarking, 37, 51
full-text searching, 157, 305–313, 479
  on BoardReader.com, 765
  Boolean full-text searches, 308
  collection, 306
  on Mininova.org, 764
  parser plugins, 297
  Sphinx storage engine, 749
functional partitioning, 531, 564
furious flushing, 49, 704, 706
Fusion-io, 407

G

Galbraith, Patrick, 296

Galera, 549, 577, 579
Ganglia, 670
garbage collection, 401
GDB stack traces, 99
gdb tool, 99–100
general log, 81
GenieDB, 549, 551
Gentoo, 683
GEOMETRY type, 157
geospatial searches, 25, 157, 262
GET_LOCK() function, 256, 288
get_name_from_id() function, 613
Gladwell, Malcom, 571
glibc libraries, 348
global locks, 736, 738
global scope, 333
global version/session splits, 558
globally unique IDs (GUIDs), 545
gnuplot, 49, 96
Goal (Goldratt), 526, 565
Goal-Driven Performance Optimization white paper, 70
GoldenGate, Oracle, 516
Goldratt, Eliyahu M., 526, 565
Golubchik, Sergei, 298
Graphite, 670
great-circle formula, 259
GREATEST() function, 254
grep, 638
Grimmer, Lenz, 659
Groonga storage engine, 23
Groundwork Open Source, 669
GROUP BY queries, 135, 137, 163, 244, 312, 752
group commit, 314
Grouply.com, 769
GROUP_CONCAT() function, 230
Guerrilla Capacity Planning (Gunther), 525, 565
GUID values, 545
Gunther, Neil J., 525, 565
gunzip tool, 716
gzip compression, 609, 716, 718

H

Hadoop, 620
handler API, 228
handler operations, 228, 265, 690
HandlerSocket, 618
HAProxy, 556
hard disks, choosing, 398
hardware and software RAID, 418
hardware threads, 388
hash codes, 152
hash indexes, 21, 152
hash joins, 234
Haversine formula, 259
header, 693
headroom, 573
HEAP tables, 20
heartbeat record, 487
HEX() function, 130
Hibernate Core interfaces, 547
Hibernate Shards, 547
high availability
  achieving, 569–572
  avoiding single points of failure, 572–581
  defined, 567
  failover and failback, 581–585
High Availability Linux project, 582
high bits, 506
High Performance Web Sites (Souders), 608
high throughput, 389
HIGH_PRIORITY hint, 238
hit rate, 322
HiveDB, 547
hot data, segregating, 269
"hot" online backups, 17
How Complex Systems Fail (Cook), 571
HTTP proxy, 585
http_load tool, 51, 54
Hutchings, Andrew, 298
Hyperic HQ, 669
hyperthreading, 389

I

I/O
  benchmark, 57
  InnoDB, 357–363
  MyISAM, 369–371
  performance, 595
  slave thread, 450
I/O-bound machines, 443
IaaS (Infrastructure as a Service), 589
.ibd files, 356, 366, 648
Icinga, 668
id column, 723
identifiers, choosing, 129–131

idle machine's vmstat output, 444
IF() function, 254
IfP (instrumentation-for-php), 78
IGNORE INDEX hint, 165, 240
implicit locking, 11
IN() function, 190–193, 219, 260
incr() function, 546
incremental backups, 630
.index files, 464
index-covered queries, 178–181
indexer, Sphinx, 756
indexes
  benefits of, 158
  case study, 189–194
  clustered, 168–176
  covering, 177–182
  and locking, 188
  maintaining, 194–198
  merge optimizations, 234
  and mismatched PARTITION BY, 270
  MyISAM storage engine, 143
  order of columns, 165–168
  packed (prefix-compressed), 184
  reducing fragmentation, 197
  redundant and duplicate, 185–187
  and scans, 182–184, 269
  statistics, 195, 220
  strategies for high performance, 159–168
  types of, 148–158
  unused, 187
INET_ATON() function, 131
INET_NTOA() function, 131
InfiniDB, Calpont, 23
info() function, 195
Infobright, 22, 28, 117, 269
INFORMATION_SCHEMA tables, 14, 110, 297, 499, 742–744
infrastructure, 617
Infrastructure as a Service (IaaS), 589
Ingo, Henrik, 515, 683
inner joins, 216
Innobase Oy, 30
InnoDB, 13, 15
  advanced settings, 383–385
  buffer pool, 349, 711
  concurrency configuration, 372
  crash recovery, 655–658
  data dictionary, 356, 711
  data layout, 172–176
  Data Recovery Toolkit, 195
  and deadlocks, 9
  and filesystem snapshots, 644–646
  flushing algorithm, 412
  Hot Backup, 457, 658
  I/O configuration, 357–363, 411
  lock waits in, 740–744
  log files, 411
  and query cache, 326
  release history, 16
  row locks, 188
  tables, 710, 742
  tablespace, 364
  transaction log, 357, 496
InnoDB locking selects, 503
innodb variable, 383
InnoDB-specific variables, 692
innodb_adaptive_checkpoint variable, 412
innodb_analyze_is_persistent variable, 197, 356
innodb_autoinc_lock_mode variable, 177, 384
innodb_buffer_pool_instances variable, 384
innodb_buffer_pool_size variable, 348
innodb_commit_concurrency variable, 373
innodb_concurrency_tickets variable, 373
innodb_data_file_path variable, 364
innodb_data_home_dir variable, 364
innodb_doublewrite variable, 368
innodb_file_io_threads variable, 702
innodb_file_per_table variable, 344, 362, 365, 414, 419, 648, 658
innodb_flush_log_at_trx_commit variable, 360, 364, 369, 418, 491, 508
innodb_flush_method variable, 344, 361, 419, 437
innodb_flush_neighbor_pages variable, 412
innodb_force_recovery variable, 195, 657
innodb_io_capacity variable, 384, 411
innodb_lazy_drop_table variable, 366
innodb_locks_unsafe_for_binlog variable, 505, 508
innodb_log_buffer_size variable, 359
innodb_log_files_in_group variable, 358
innodb_log_file_size variable, 358
innodb_max_dirty_pages_pct variable, 350
innodb_max_purge_lag variable, 367
innodb_old_blocks_time variable, 385
innodb_open_files variable, 356

innodb_overwrite_relay_log_info variable, 383
innodb_read_io_threads variable, 385, 702
innodb_recovery_stats variable, 359
innodb_stats_auto_update variable, 197
innodb_stats_on_metadata variable, 197, 356
innodb_stats_sample_pages variable, 196
innodb_strict_mode variable, 385
innodb_support_xa variable, 314, 330
innodb_sync_spin_loops variable, 695
innodb_thread_concurrency variable, 101, 372
innodb_thread_sleep_delay variable, 372
innodb_use_sys_stats_table variable, 197, 356
innodb_version variable, 742
innodb_write_io_threads variable, 385, 702
innotop tool, 500, 672, 693
INSERT ... SELECT statements, 28, 240, 488, 503
insert buffer, 413, 703
INSERT command, 267, 278
INSERT ON DUPLICATE KEY UPDATE command, 252, 682
insert-to-select rate, 323
inspecting server status variables, 346
INSTEAD OF trigger, 278
instrumentation, 73
instrumentation-for-php (IfP), 78
INT type, 117
integer computations, 117
integer types, 117, 130
Intel X-25E drives, 404
Intel Xeon X5670 Nehalem CPU, 598
interface tools, 665
intermittent problems, diagnosing, 92
  capturing diagnostic data, 97–102
  case study, 102–110
  single-query versus server-wide problems, 93–96
internal concurrency issues, 391
internal XA transactions, 314
intra-row fragmentation, 198
introducers, 300
invalidation on read, 615
ionice, 626
iostat, 438–442, 591, 646
IP addresses, 560, 584
IP takeover, 583
ISNULL() function, 254
isolating columns, 159
isolation, 7
iterative optimization by benchmarking, 338

J

JMeter, 51
joins, 132, 234
  decomposition, 209
  execution strategy, 220
  JOIN queries, 244
  optimizers for, 223–226
journaling filesystems, 433
Joyent, 589

K

Karlsson, Anders, 510
Karwin, Bill, 256
Keep-Alive, 608
key block size, 353
key buffers, 351
key column, 729
key_buffer_size variable, 335, 351
key_len column, 729
Köhntopp, Kristian, 252
Kyte, Tom, 76

L

L-values, 250
lag, 484, 486, 507–511
Lahdenmaki, Tapio, 158, 204
LAST() function, 255
LAST_INSERT_ID() function, 239
latency, 38, 398, 576
LATEST DETECTED DEADLOCK, 697
LATEST FOREIGN KEY ERROR, 695
Launchpad, 64
lazy UNIONs, 254
LDAP authentication, 298
Leach, Mike, 158, 204
LEAST() function, 254
LEFT JOIN queries, 219
LEFT OUTER JOIN queries, 231
left-deep trees, 223
Leith, Mark, 712
LENGTH() function, 254, 304
lighttpd, 608
lightweight profiling, 76
LIMIT query, 218, 227, 246

limited replication bandwidth, 511
linear scalability, 524
"lint checking", 249
Linux Virtual Server (LVS), 449, 556, 560
Linux-HA stack, 582
linuxthreads, 435
Little's Law, 441
load balancers, 561
load balancing, 449, 555–565
LOAD DATA FROM MASTER command, 457
LOAD DATA INFILE command, 79, 301, 504, 508, 511, 600, 651
LOAD INDEX command, 272, 352
LOAD TABLE FROM MASTER command, 457
LOAD_FILE() function, 281
local caches, 612
local shared-memory caches, 613
locality of reference, 393
lock contention, 503
LOCK IN SHARE MODE command, 240
LOCK TABLES command, 11, 632
lock time, 626
lock waits, 735, 740–744
lock-all-tables variable, 457
lock-free InnoDB backups, 644
locks
  debugging, 735–744
  granularities, 4
  implicit and explicit, 11
  read/write, 4
  row, 5
  table, 5
log buffer, 358–361
log file coordinates, 456
log file size, 344, 358–361, 411
log positions, locating, 492
log servers, 481, 654
log threads, 702
log, InnoDB transaction, 703
logging, 10, 25
logical backups, 627, 637–639, 649–651
logical concurrency issues, 391
logical reads, 395
logical replication, 460
logical unit numbers (LUNs), 423
log_bin variable, 458
log_slave_updates variable, 453, 465, 468, 511, 635
LONGBLOB type, 122
LONGTEXT type, 122
lookup tables, 20
loose index scans, 235
lost time, 74
low latency, 389
LOW_PRIORITY hint, 238
Lua language, 53
Lucene, 313
LucidDB, 23
LUNs (logical unit numbers), 423
LVM snapshots, 434, 633, 640–648
lvremove command, 643
LVS (Linux Virtual Server), 449, 556, 560
lzo, 626

M

Maatkit (see Percona Toolkit)
maintenance operations, 271
malloc() function, 319
manual joins, 606
mapping tables, 20
MariaDB, 19, 484, 681
master and replicas, 468, 474, 564
master shutdown, unexpected, 495
master-data variable, 457
master-master in active-active mode, 469
master-master in active-passive mode, 471
master-master replication, 473, 505
master.info file, 459, 464, 489, 496
Master_Log_File, 491
MASTER_POS_WAIT() function, 495, 564
MATCH() function, 216, 306, 307, 311
materialized views, 138, 280
Matsunobu, Yoshinori, 581
MAX() function, 217, 237, 292
Maxia, Giuseppe, 282, 456, 512, 515, 518, 667
maximum system capacity, 521, 609
max_allowed_packet variable, 381
max_connections setting variable, 378
max_connect_errors variable, 381
max_heap_table_size setting variable, 378
mbox mailbox messages, 3
MBRCONTAINS() function, 157
McCullagh, Paul, 22
MD5() function, 53, 130, 156, 507

md5sum, 718
mean time between failures (MTBF), 569
mean time to recover (MTTR), 569–572, 576, 582, 586
measurement uncertainty, 72
MEDIUMBLOB type, 122
MEDIUMINT type, 117
MEDIUMTEXT type, 122
memcached, 533, 546, 613, 616
Memcached Access, 618
memory
  allocating for caches, 349
  configuring, 347–356
  consumption formula for, 341
  InnoDB buffer pool, 349
  InnoDB data dictionary, 356
  limits on, 347
  memory-to-disk ratio, 397
  MyISAM key cache, 351–353
  per-connection needs, 348
  pool, 704
  reserving for operating system, 349
  size, 595
  Sphinx RAM, 751
  table cache, 354
  thread cache, 353
Memory storage engine, 20
Merge storage engine, 21
merge tables, 273–276
merged read and write requests, 440
mget() call, 616
MHA toolkit, 581
middleman solutions, 560–563, 584
migration, benchmarking after, 46
Millsap, Cary, 70, 74, 341
MIN() function, 217, 237, 292
Mininova.org, 764
mk-parallel-dump tool, 638
mk-parallel-restore tool, 638
mk-query-digest tool, 72
mk-slave-prefetch tool, 510
MLC (multi-level cell), 402, 407
MMM replication manager, 572, 580
mod_log_config variable, 79
MonetDB, 23
Monitis, 671
monitoring tools, 667–676
MONyog, 671
mpstat tool, 438
MRTG (Multi Router Traffic Grapher), 430, 669
MTBF (mean time between failures), 569
mtop tool, 672
MTTR (mean time to recovery), 569–572, 576, 582, 586
Mulcahy, Lachlan, 659
Multi Router Traffic Grapher (MRTG), 430, 669
multi-level cell (MLC), 402, 407
multi-query mechanism, 753
multicolumn indexes, 163
multiple disk volumes, 427
multiple partitioning keys, 537
multisource replication, 470, 480
multivalued attributes, 757, 761
Munin, 670
MVCC (multiversion concurrency control), 12, 551
my.cnf file, 452, 490, 501
.MYD file, 371, 633, 648
mydumper, 638, 659
.MYI file, 633, 648
MyISAM storage engine, 17
  and backups, 631
  concurrency configuration, 18, 373
  and COUNT() queries, 242
  data layout, 171
  delayed key writes, 19
  indexes, 18, 143
  key block size, 353
  key buffer/cache, 351–353, 690
  performance, 19
  tables, 19, 498
myisamchk, 629
myisampack, 276
mylvmbackup, 658, 659
MySQL
  concurrency, 371–374
  configuration mechanisms, 332–337
  development model, 33
  GPL-licensing, 33
  logical architecture, 1
  proprietary plugins, 33
  Sandbox script, 456, 481
  version history, 29–33, 182, 188
MySQL 5.1 Plugin Development (Golubchik & Hutchings), 298
MySQL Benchmark Suite, 52, 55

MySQL Cluster, 577
MySQL Enterprise Backup, 457, 624, 627, 631, 658
MySQL Enterprise Monitor, 80, 670
MySQL Forge, 667, 710
MySQL High Availability (Bell et al.), 519
MySQL Stored Procedure Programming (Harrison & Feuerstein), 282
MySQL Workbench Utilities, 665
mysql-bin.index file, 464
mysql-relay-bin.index file, 464
mysqladmin, 666, 686
mysqlbinlog tool, 460, 481, 492, 654
mysqlcheck tool, 629, 666
mysqld tool, 99, 344
mysqldump tool, 456, 488, 623, 627, 637, 660
mysqlhotcopy tool, 658
mysqlimport tool, 627, 651
mysqlslap tool, 51
mysql_query() function, 212, 292
mysql_unbuffered_query() function, 212
mytop tool, 672

N

Nagios, 668
Nagios System and Network Monitoring (Barth), 643, 668
name locks, 736, 739
NAS (network-attached storage), 422–427
NAT (network address translation), 584
Native POSIX Threads Library (NPTL), 435
natural identifiers, 134
natural-language full-text searches, 306
NDB API, 619
NDB Cluster storage engine, 21, 535, 549, 550, 576
nesting cursors, 290
netcat, 717
network address translation (NAT), 584
network configuration, 429–431
network overhead, 202
network performance, 595
network provider, reliance on single, 572
network-attached storage (NAS), 422–427
New Relic, 77, 671
next-key locking, 17
NFS, SAN over, 424
Nginx, 608, 612
nice, 626
nines rule of availability, 567
Noach, Shlomi, 187, 666, 687, 710
nodes, 531, 538
non-SELECT queries, 721
nondeterministic statements, 499
nonrepeatable reads, 8
nonreplicated data, 501
nonsharded data, 538
nontransactional tables, 498
nonunique server IDs, 500
nonvolatile random access memory (NVRAM), 400
normalization, 133–136
NOT EXISTS() queries, 219, 232
NOT NULL, 116, 682
NOW() function, 316
NOW_USEC() function, 296, 513
NPTL (Native POSIX Threads Library), 435
NULL, 116, 133, 270
null hypothesis, 47
NULLIF() function, 254
NuoDB, 22
NVRAM (nonvolatile random access memory), 400

O

object versioning, 615
object-relational mapping (ORM) tool, 131, 148, 606
OCZ, 407
OFFSET variable, 246
OLTP (online transaction processing), 22, 38, 59, 478, 509, 596
on-controller cache (see RAID)
on-disk caches, 614
on-disk temporary tables, 122
online transaction processing (OLTP), 22, 38, 59, 478, 509, 596
open() function, 363
openark kit, 666
opened tables, 355
opening and locking partitions, 271
OpenNMS, 669
operating system
  choosing an, 431
  how to select CPUs for MySQL, 388
  optimization, 387
  status of, 438–444
  what limits performance, 387

oprofile tool, 99–102, 111
Opsview, 668
optimistic concurrency control, 12
optimization, 3
(see also application-level optimization)
(see also query optimization)
BLOB workload, 375
DISTINCT queries, 244
filesort, 377
full-text indexes, 312
GROUP BY queries, 244, 752, 768
JOIN queries, 244
LIMIT and OFFSET, 246
OPTIMIZE TABLE command, 170, 310, 501
optimizer traces, 734
optimizer_prune_level, 240
optimizer_search_depth, 240
optimizer_switch, 241
prepared statements, 292
queries, 272
query cache, 327
query optimizer, 215–220
RAID performance, 415–417
ranking queries, 250
selects on Sahibinden.com, 767
server setting optimization, 331
sharded JOIN queries on Grouply.com, 769
for solid-state storage, 410–414
sorts, 193
SQL_CALC_FOUND_ROWS variable, 248
subqueries, 244
TEXT workload, 375
through profiling, 72–75, 91
UNION variable, 248
Optimizer
hints
DELAYED, 239
FOR UPDATE, 240
FORCE INDEX, 240
HIGH_PRIORITY, 238
IGNORE INDEX, 240
LOCK IN SHARE MODE, 240
LOW_PRIORITY, 238
SQL_BIG_RESULT, 239
SQL_BUFFER_RESULT, 239
SQL_CACHE, 239
SQL_CALC_FOUND_ROWS, 239
SQL_NO_CACHE, 239
SQL_SMALL_RESULT, 239
STRAIGHT_JOIN, 239
USE INDEX, 240
limitations of
correlated subqueries, 229–233
equality propogation, 234
hash joins, 234
index merge optimizations, 234
loose index scans, 235
MIN() and MAX(), 237
parallel execution, 234
SELECT and UPDATE on the Same Table, 237
UNION limitations, 233
query, 214–227
complex queries versus many queries, 207
COUNT() aggregate function, 241
join decomposition, 209
limitations of MySQL, 229–238
optimizing data access, 202–207
reasons for slow queries, 201
restructuring queries, 207–209
Optimizing Oracle Performance (Millsap), 70, 341
options, 332
OQGraph storage engine, 23
Oracle Database, 408
Oracle development milestones, 33
Oracle Enterprise Linux, 432
Oracle GoldenGate, 516
ORDER BY queries, 163, 182, 226, 253
order processing, 26
ORM (object-relational mapping), 148, 606
OurDelta, 683
out-of-sync replicas, 488
OUTER JOIN queries, 221
outer joins, 216
outliers, 74
oversized packets, 511
O_DIRECT variable, 362
O_DSYNC variable, 363

P

Pacemaker, 560, 582
packed indexes, 184
packed tables, 19

PACK_KEYS variable, 184
page splits, 170
paging, 436
PAM authentication, 298
parallel execution, 234
parallel result sets, 753
parse tree, 3
parser, 214
PARTITION BY variable, 265, 270
partitioning, 415
  across multiple nodes, 531
  how to use, 268
  keys, 535
  with replication filters, 564
  sharding, 533–547, 565, 755
  tables, 265–276, 329
  types of, 267
passive caches, 611
Patricia tries, 158
PBXT, 22
PCIe cards, 400, 406
Pen, 556
per-connection memory needs, 348
per-connection needs, 348
percent() function, 676
percentile response times, 38
Percona InnoDB Recovery Toolkit, 657
Percona Server, 598, 679, 711
  BLOB and TEXT types, 122
  buffer pool, 711
  bypassing operating system caches, 344
  corrupted tables, 657
  doublewrite buffer, 411
  enhanced slow query log, 89
  expand_fast_index_creation, 198
  extended slow query log, 323, 330
  fast warmup features, 351, 563, 598
  FNV64() function, 157
  HandlerSocket plugin, 297
  idle transaction timeout parameter, 744
  INFORMATION_SCHEMA.INDEX_STATISTICS table, 187
  innobd_use_sys_stats_table option, 197
  InnoDB online text creation, 144
  innodb_overwrite_relay_log_info option, 383
  innodb_read_io_threads option, 702
  innodb_recovery_stats option, 359
  innodb_use_sys_stats_table option, 356
  innodb_write_io_threads option, 702
  larger log files, 411
  lazy page invalidation, 366
  limit data dictionary size, 356, 711
  mutex issues, 384
  mysqldump, 628
  object-level usage statistics, 110
  query-level instrumentation, 73
  read-ahead, 412
  replication, 484, 496, 508, 516
  slow query log, 74, 80, 84, 89, 95
  stripping query comments, 316
  temporary tables, 689, 711
  user statistics tables, 711
Percona Toolkit, 666
  Aspersa, 666
  Maatkit, 658, 666
  mk-parallel-dump tool, 638
  mk-parallel-restore tool, 638
  mk-query-digest tool, 72
  mk-slave-prefetch tool, 510
  pt-archiver, 208, 479, 504, 545, 553
  pt-collect, 99, 442
  pt-deadlock-logger, 697
  pt-diskstats, 45, 442
  pt-duplicate-key-checker, 187
  pt-fifo-split, 651
  pt-find, 502
  pt-heartbeat, 476, 487, 492, 559
  pt-index-usage, 187
  pt-kill, 744
  pt-log-player, 340
  pt-mext, 347, 687
  pt-mysql-summary, 100, 103, 347, 677
  pt-online-schema-change, 29
  pt-pmp, 99, 101, 390
  pt-query-advisor, 249
  pt-query-digest, 375, 507, 563
    extracting from comments, 79
    profiling, 72–75
    query log, 82–84
    slow query logging, 90, 95, 340
  pt-sift, 100, 442
  pt-slave-delay, 516, 634
  pt-slave-restart, 496
  pt-stalk, 98, 99, 442
  pt-summary, 100, 103, 677
  pt-table-checksum, 488, 495, 519, 634
  pt-table-sync, 489

pt-tcp-model, 611
pt-upgrade, 187, 241, 570, 734
pt-visual-explain, 733
Percona tools, 52, 64–66, 195
Percona XtraBackup, 457, 624, 627, 631, 648, 658
Percona XtraDB Cluster, 516, 549, 577–580, 680
performance optimization, 69–72, 107
plotting metrics, 49
profiling, 72–75
SAN, 424
views and, 279
Performance Schema, 90
Perl scripts, 572
Perldoc, 662
perror utility, 355
persistent connections, 561, 607
persistent memory, 597
pessimistic concurrency control, 12
phantom reads, 8
PHP profiling tools, 77
phpMyAdmin tool, 666
phrase proximity ranking, 759
phrase searches, 309
physical reads, 395
physical size of disk, 399
pigz tool, 626
"pileups", 69
Pingdom, 671
pinging, 606, 689
Planet MySQL blog aggregator, 667
planned promotions, 490
plugin-specific variables, 692
plugins, 297
point-in-time recovery, 625, 652
poor man's profiler, 101
port forwarding, 584
possible_keys column, 729
post-mortems, 571
PostgreSQL, 258
potential cache size, 323
power grid, 572
preferring a join, 244
prefix indexes, 160–163
prefix-compressed indexes, 184
preforking, 608
pregenerating content, 617
prepared statements, 291–295, 329
preprocessor, 214
Preston, W. Curtis, 621
primary key, 17, 173–176
PRIMARY KEY constraint, 185
priming the cache, 509
PROCEDURE ANALYSE command, 297
procedure plugins, 297
processor speed, 392
profiling
and application speed, 76
applications, 75–80
diagnosing intermittent problems, 92–110
interpretation, 74
MySQL queries, 80–84
optimization through, 72–75, 91
single queries, 84–91
tools, 72, 110–112
promotions of replicas, 491, 583
propagation of changes, 584
proprietary plugins, 33
proxies, 556, 584, 609
pruning, 270
pt-archiver tool, 208, 479, 504, 545, 553
pt-collect tool, 99, 442
pt-deadlock-logger tool, 697
pt-diskstats tool, 45, 442
pt-duplicate-key-checker tool, 187
pt-fifo-split tool, 651
pt-find tool, 502
pt-heartbeat tool, 476, 487, 492, 559
pt-index-usage tool, 187
pt-kill tool, 744
pt-log-player tool, 340
pt-mext tool, 347, 687
pt-mysql-summary tool, 100, 103, 347, 677
pt-online-schema-change tool, 29
pt-pmp tool, 99, 101, 390
pt-query-advisor tool, 249
pt-query-digest (see Percona Toolkit)
pt-sift tool, 100, 442
pt-slave-delay tool, 516, 634
pt-slave-restart tool, 496
pt-stalk tool, 98, 99, 442
pt-summary tool, 100, 103, 677
pt-table-checksum tool, 488, 495, 519, 634
pt-table-sync tool, 489
pt-tcp-model tool, 611
pt-upgrade tool, 187, 241, 570, 734
pt-visual-explain tool, 733

PURGE MASTER LOGS command, 369, 464, 486
purging old binary logs, 636
pushdown joins, 550, 577

Q

Q mode, 673
Q4M storage engine, 23
Qcache_lowmem_prunes variable, 325
query cache, 214, 315, 330, 690
  alternatives to, 328
  configuring and maintaining, 323–325
  InnoDB and the, 326
  memory use, 318
  optimizations, 327
  when to use, 320–323
query execution
  MySQL client/server protocol, 210–213
  optimization process, 214
  query cache, 214, 315–328
query execution engine, 228
query logging, 95
query optimization, 214–227
  complex queries versus many queries, 207
  COUNT() aggregate function, 241
  join decomposition, 209
  limitations of MySQL, 229–238
  optimizing data access, 202–207
  reasons for slow queries, 201
  restructuring queries, 207–209
query states, 213
query-based splits, 557
querying across shards, 537
query_cache_limit variable, 324
query_cache_min_res_unit value variable, 324
query_cache_size variable, 324, 336
query_cache_type variable, 323
query_cache_wlock_invalidate variable, 324
queue scheduler, 434
queue tables, 256
queue time, 204
quicksort, 226

R

R-Tree indexes, 157
Rackspace Cloud, 589
RAID
  balancing hardware and software, 418
  configuration and caching, 419–422
  failure, recovery, and monitoring, 417
  moving files from flash to, 411
  not for backup, 624
  performance optimization, 415–417
  splits, 647
  with SSDs, 405
RAND() function, 160, 724
random read-ahead, 412
random versus sequential I/O, 394
RANGE COLUMNS type, 268
range conditions, 192
raw file
  backup, 627
  restoration, 648
RDBMS technology, 400
RDS (Relational Database Service), 589, 600
read buffer size, 343
READ COMMITTED isolation level, 8, 13
read locks, 4, 189
read threads, 703
READ UNCOMMITTED isolation level, 8, 13
read-ahead, 412
read-around writes, 353
read-mostly tables, 26
read-only variable, 26, 382, 459, 479
read-write splitting, 557
read_buffer_size variable, 336
Read_Master_Log_Pos, 491
read_rnd_buffer_size variable, 336
real number data types, 118
rebalancing shards, 544
records_in_range() function, 195
recovery
  from a backup, 647–658
  defined, 622
  defining requirements, 623
  more advanced techniques, 653
recovery point objective (RPO), 623, 625
recovery time objective (RTO), 623, 625
Red Hat, 432, 683
Redis, 620
redundancy, replication-based, 580
Redundant Array of Inexpensive Disks (see RAID)
redundant indexes, 185–187
ref column, 730

Relational Database Index Design and the Optimizers (Lahdenmaki & Leach), 158, 204
Relational Database Service (RDS), Amazon, 589, 600
relay log, 450, 496
relay-log.info file, 464
relay_log variable, 453, 459
relay_log_purge variable, 459
relay_log_space_limit variable, 459, 511
RELEASE_LOCK() function, 256
reordering joins, 216
REORGANIZE PARTITION command, 271
REPAIR TABLE command, 144, 371
repairing MyISAM tables, 18
REPEATABLE READ isolation level, 8, 13, 632
replica hardware, 414
replica shutdown, unexpected, 496
replicate_ignore_db variable, 478
replication, 447, 634
  administration and maintenance, 485
  advanced features in MySQL, 514
  backing up configuration, 630
  and capacity planning, 482–485
  changing masters, 489–494
  checking consistency of, 487
  checking for up-to-dateness, 565
  configuring master and replica, 452
  creating accounts for, 451
  custom solutions, 477–482
  filtering, 466, 564
  how it works, 449
  initializing replica from another server, 456
  limitations, 512
  master and multiple replicas, 468
  master, distribution master, and replicas, 474
  master-master in active-active mode, 469
  master-master in active-passive mode, 471
  master-master with replicas, 473
  measuring lag, 486
  monitoring, 485
  other technologies, 516
  problems and solutions, 495–512
  problems solved by, 448
  promotions of replicas, 491, 583
  recommended configuration, 458
  replica consistency with master, 487
  replication files, 463
  resyncing replica from master, 488
  ring, 473
  row-based, 447, 460–463
  sending events to other replicas, 465
  setting up, 451
  speed of, 512–514
  splitting reads and writes in, 557
  starting the replica, 453–456
  statement-based, 447, 460–463
  status, 708
  switching master-master configuration roles, 494
  topologies, 468, 490
  tree or pyramid, 476
REPLICATION CLIENT privilege, 452
REPLICATION SLAVE privilege, 452
replication-based redundancy, 580
RESET QUERY CACHE command, 325
RESET SLAVE command, 490
resource consumption, 70
response time, 38, 69, 204
restoring
  defined, 622
  logical backups, 649–651
RethinkDB, 22
ring replication, 473
ROLLBACK command, 499
round-robin database (RRD) files, 669
row fragmentation, 198
row locks, 5, 12
ROW OPERATIONS, 705
row-based logging, 636
row-based replication, 447, 460–463
rows column, 731
rows examined, number of, 205
rows returned, number of, 205
ROW_COUNT command, 287
RPO (recovery point objective), 623, 625
RRDTool, 669
rsync, 195, 456, 717, 718
RTO (recovery time objective), 623, 625
running totals and averages, 255

S

safety and sanity settings, 380–383
Sahibinden.com, 767
SandForce, 407
SANs (storage area networks), 422–427

sar, 438
sargs, 166
SATA SSDs, 405
scalability, 521
  by clustering, 548
  by consolidation, 547
  frequency, 392
  and load balancing, 555
  mathematical definition, 523
  multiple CPUs/cores, 391
  planning for, 527
  preparing for, 528
  "scale-out" architecture, 447
  scaling back, 552
  scaling out, 531–547
  scaling pattern, 391
  scaling up, 529
  scaling writes, 483
  Sphinx, 754
  universal law of, 525–527
scalability measurements, 39
ScaleArc, 547, 549
ScaleBase, 547, 549, 551, 594
ScaleDB, 407, 574
scanning data, 269
scheduled tasks, 504
schemas, 13
  changes, 29
  design, 131
  normalized and denormalized, 135
Schooner Active Cluster, 549
scope, 333
scp, 716
search engine, selecting the right, 24–28
search space, 226
searchd, Sphinx, 746, 754, 756–766
secondary indexes, 17, 656
security, connection management, 2
sed, 638
segmented key cache, 19
segregating hot data, 269
SELECT command, 237, 267, 721
SELECT FOR UPDATE command, 256, 287
SELECT INTO OUTFILE command, 301, 504, 508, 600, 638, 651, 657
SELECT types, 690
selective replication, 477
selectivity, index, 160
select_type column, 724
SEMAPHORES, 693
sequential versus random I/O, 394
sequential writes, 576
SERIALIZABLE isolation level, 8, 13
serialized writes, 509
server, 685
  adding/removing, 563
  configuration, backing up, 630
  consolidation, 425
  INFORMATION_SCHEMA database, 711
  MySQL configuration, 332
  PERFORMANCE_SCHEMA database, 712
  profiling and speed of, 76, 80
  server-wide problems, 93–96
  setting optimization, 331
  SHOW ENGINE INNODB MUTEX command, 707–709
  SHOW ENGINE INNODB STATUS command, 692–706
  SHOW PROCESSLIST command, 706
  SHOW STATUS command, 686–692
  status variables, 346
  workload profiling, 80
server-side prepared statements, 295
service time, 204
session scope, 333
session-based splits, 558
SET CHARACTER SET command, 300
SET GLOBAL command, 494
SET GLOBAL SQL_SLAVE_SKIP_COUNTER command, 654
SET NAMES command, 300
SET NAMES utf8 command, 300, 606
SET SQL_LOG_BIN command, 503
SET TIMESTAMP command, 635
SET TRANSACTION ISOLATION LEVEL command, 11
SET type, 128, 130
SetLimits() function, 748, 764
SetMaxQueryTime() function, 764
SeveralNines, 550, 577
SHA1() function, 53, 130, 156
Shard-Query system, 547
sharding, 533–547, 565, 755
shared locks, 4
shared storage, 573–576
SHOW BINLOG EVENTS command, 486, 708

SHOW commands, 255
SHOW CREATE TABLE command, 117, 163
SHOW CREATE VIEW command, 280
SHOW ENGINE INNODB MUTEX command, 695, 707–709
SHOW ENGINE INNODB STATUS command, 97, 359, 366, 384, 633, 692–706, 740
SHOW FULL PROCESSLIST command, 81, 700
SHOW GLOBAL STATUS command, 88, 93, 346, 686
SHOW INDEX command, 197
SHOW INDEX FROM command, 196
SHOW INNODB STATUS command (see SHOW ENGINE INNODB STATUS command)
SHOW MASTER STATUS command, 452, 457, 486, 490, 558, 630, 643
SHOW PROCESSLIST command, 94–96, 256, 289, 606, 706
SHOW PROFILE command, 85–89
SHOW RELAYLOG EVENTS command, 708
SHOW SLAVE STATUS command, 453, 457, 486, 491, 558, 630, 708
SHOW STATUS command, 88, 352
SHOW TABLE STATUS command, 14, 197, 365, 672
SHOW VARIABLES command, 352, 685
SHOW WARNINGS command, 222, 277
signed types, 117
single-component benchmarking, 37, 51
single-level cell (SLC), 402, 407
single-shard queries, 535
single-transaction variable, 457, 632
skip_innodb variable, 476
skip_name_resolve variable, 381, 429, 570
skip_slave_start variable, 382, 459
slavereadahead tool, 510
slave_compressed_protocol variable, 475, 511
slave_master_info variable, 383
slave_net_timeout variable, 382
Slave_open_temp_tables variable, 503
SLC (single-level cell), 402, 407
Sleep state, 607
SLEEP() function, 256, 682, 737
sleeping before entering queue, 373
slots, 694
slow queries, 71, 74, 80, 89, 109, 321
SMALLBLOB type, 122
SMALLINT type, 117
SMALLTEXT type, 122
Smokeping tool, 430
snapshots, 457, 624, 640–648
Solaris SPARC hardware, 431
Solaris ZFS filesystem, 431
solid-state drives (SSD), 147, 268, 361, 404
solid-state storage, 400–414
sort buffer size, 343
sort optimizations, 226, 691
sorting, 193
sort_buffer_size variable, 336
Souders, Steve, 608
SourceForge, 52
SPARC hardware, 431
spatial indexes, 157
Sphinx, 313, 619, 745, 770
  advanced performance control, 763
  applying WHERE clauses, 750
  architectural overview, 756–758
  efficient and scalable full-text searching, 749
  filtering, 761
  finding top results in order, 751
  geospatial search functions, 262
  installation overview, 757
  optimizing GROUP BY queries, 752, 768
  optimizing selects on Sahibinden.com, 767
  optimizing sharded JOIN queries on Grouply.com, 769
  phrase proximity ranking, 759
  searching, 746–748
  special features, 759–764
  SphinxSE, 756, 759, 761, 767
  support for attributes, 760
  typical partition use, 758
Spider storage engine, 24
spin-wait, 695
spindle rotation speed, 399
splintering, 533–547
split-brain syndrome, 575, 578
splitting reads and write in replication, 557
Splunk, 671
spoon-feeding, 608
SQL and Relational Theory (Date), 255
SQL Antipatterns (Karwin), 256
SQL dumps, 637
SQL interface prepared statements, 295

SQL slave thread, 450
SQL statements, 638
SQL utilities, 667
sql-bench, 52
SQLyog tool, 665
SQL_BIG_RESULT hint, 239, 245
SQL_BUFFER_RESULT hint, 239
SQL_CACHE hint, 239
SQL_CACHE variable, 321, 328
SQL_CALC_FOUND_ROWS hint, 239
SQL_CALC_FOUND_ROWS variable, 248
sql_mode, 382
SQL_MODE configuration variable, 245
SQL_NO_CACHE hint, 239
SQL_NO_CACHE variable, 328
SQL_SMALL_RESULT hint, 239, 245
Squid, 608
SSD (solid-state drives), 147, 268, 361, 404
SSH, 716
staggering numbers, 505
stale-data splits, 557
"stalls", 69
Starkey, Jim, 22
START SLAVE command, 654
START SLAVE UNTIL command, 654
start-position variable, 498
statement handles, 291
statement-based replication, 447, 460–463
static optimizations, 216
static query analysis, 249
STEC, 407
STONITH, 584
STOP SLAVE command, 487, 490, 498
stopwords, 306, 312
storage area networks (SANs), 422–427
storage capacity, 399
storage consolidation, 425
storage engine API, 2
storage engines, 13, 23–28
  Archive, 19
  Blackhole, 20
  column-oriented, 22
  community, 23
  and consistency, 633
  CSV, 20
  Falcon, 22
  Federated, 20
  InnoDB, 15
  Memory, 20
  Merge, 21
  mixing, 11, 500
  MyISAM, 18
  NDB Cluster, 21
  OLTP, 22
  ScaleDB, 574
  XtraDB, 680
stored code, 282–284, 289
Stored Procedure Library, 667
stored procedures and functions, 284
stored routines, 282, 329
strace tool, 99, 111
STRAIGHT_JOIN hint, 224, 239
string data types, 119–125, 130
string locks, 736
stripe chunk size, 420
subqueries, 218, 244
SUBSTRING() function, 122, 304, 375
sudo rules, 630
SUM() function, 139
summary tables, 136
Super Smack, 52
surrogate keys, 173
Swanhart, Justin, 138, 280, 547
swapping, 436, 444
switchover, 582
synchronization, two-way, 287
synchronous MySQL replication, 576–580
sync_relay_log variable, 383
sync_relay_log_info variable, 383
sysbench, 39, 53, 56–61, 419, 426, 598
SYSDATE() function, 382
sysdate_is_now variable, 382
system of record approach, 517
system performance, benchmarking, 44
system under test (SUT), 44
system variables, 685

T

table definition cache, 356
tables
  building a queue, 256
  cache memory, 354
  column, 724–727
  conversions, 28
  derived, 238, 277, 725
  finding and repairing corruption, 194
  INFORMATION_SCHEMA in Percona Server, 711

locks, 5, 692, 735–738
maintenance, 194–198
merge, 273–276
partitioned, 265–276, 329
reducing to an MD5 hash value, 255
SELECT and UPDATE on, 237
SHOW TABLE STATUS output, 14
splitting, 554
statistics, 220
tablespaces, 16, 364
views, 276–280
table_cache_size variable, 335, 379
tagged cache, 615
TCP, 556, 583
tcpdump tool, 81, 95, 99
tcp_max_syn_backlog variable, 430
temporal computations, 125
temporary files and tables, 21, 502, 689, 711
TEMPTABLE algorithm, 277
Texas Memory Systems, 407
TEXT type, 21, 121, 122
TEXT workload, optimizing for, 375
Theory of Constraints, 526
third-party storage engines, 21
thread and connection statistics, 688
thread cache memory, 353
threaded discussion forums, 27
threading, 213, 435
Threads_connected variable, 354, 596
Threads_created variable, 354
Threads_running variable, 596
thread_cache_size variable, 335, 354, 379
throttling variables, 627
throughput, 38, 70, 398, 576
tickets, 373
time to live (TTL), 614
time-based data partitioning, 554
TIMESTAMP type, 117, 126, 631
TIMESTAMPDIFF() function, 513
TINYBLOB type, 122
TINYINT type, 117
TINYTEXT type, 122
Tkachenko, Vadim, 405
tmp_table_size setting, 378
TokuDB, 22, 158
TO_DAYS() function, 268
TPC Benchmarks
dbt2, 61
TPC-C, 52
TPC-H, 41
TPCC-MySQL tool, 52, 64–66
transactional tables, 499
transactions, 24
ACID test, 6
deadlocks, 9
InnoDB, 366, 699
isolation levels, 7
logging, 10
in MySQL, 10
and storage engines, 24
transfer speed, 398
transferring large files, 715–718
transparency, 556, 578, 611
tree or pyramid replication, 476
tree-formatted output, 733
trial-and-error troubleshooting, 92
triggers, 97, 282, 286
TRIM command, 404
Trudeau, Yves, 262
tsql2mysql tool, 282
TTL (time to live), 614
tunefs, 433
Tungsten Replicator, Continuent, 481, 516
"tuning", 340
turbo boost technology, 392
type column, 727

U

Ubuntu, 683
UDF Library, 667
UDFs, 262, 295
unarchiving, 553
uncommitted data, 8
uncompressed files, 715
undefined server IDs, 501
underutilization, 485
UNHEX() function, 130
UNION ALL query, 248
UNION limitations, 233
UNION query, 220, 248, 254, 724–727
UNION syntax, 274
UNIQUE constraint, 185
unit of sharding, 535
Universal Scalability Law (USL), 525–527
Unix, 332, 432, 504, 582, 630
UNIX_TIMESTAMP() function, 126
UNLOCK TABLES command, 12, 142, 643
UNSIGNED attribute, 117

"unsinkable" systems, 573
unused indexes, 187
unwrapping, 255
updatable views, 278
UPDATE command, 237, 267, 278
UPDATE RETURNING command, 252
upgrades
  replication before, 449
  validating MySQL, 241
USE INDEX hint, 240
user logs, 740
user optimization issues, 39, 166
user statistics tables, 711
user-defined functions (UDFs), 262, 295
user-defined variables, 249–255
USER_STATISTICS tables, 110
"Using filesort" value, 733
"Using index" value, 733
USING query, 218
"Using temporary" value, 733
"Using where" value, 733
USL (Universal Scalability Law), 525–527
UTF-8, 298, 303
utilities, SQL, 667
UUID() function, 130, 507, 546
UUID_SHORT() function, 546

V

Valgrind, 78
validating MySQL upgrades, 241
VARCHAR type, 119, 124, 131, 513
variables, 332
  assignments in statements, 255
  setting dynamically, 335–337
  user-defined, 249–255
version-based splits, 558
versions
  and full-text searching, 310
  history of MySQL, 29–33
  improvements in MySQL 5.6, 734
  old row, 366
  replication before upgrading, 449
  version-specific comments, 289
vgdisplay command, 642
views, 276–280, 329
Violin Memory, 407
Virident, 403, 409
virtual IP addresses, 560, 583
virtualization, 548
vmstat tool, 436, 438, 442, 591, 646
volatile memory, 597
VoltDB, 549
volume groups, 641
VPForMySQL storage engine, 24

W

Wackamole, 556
waiters flag, 694
warmup, 351, 573
wear leveling, 401
What the Dog Saw (Gladwell), 571
WHERE clauses, 255, 750
whole number data types, 117
Widenius, Monty, 679, 681
Windows, 504
WITH ROLLUP variable, 246
Workbench Utilities, MySQL, 665, 666
working concurrency, 39
working sets of data, 395, 597
workload-based configuration, 375–377
worst-case selectivity, 162
write amplification, 401
write cache and power failure, 405
write locks, 4, 189
write synchronization, 565
write threads, 703
write-ahead logging, 10, 395
write-invalidate policy, 614
write-set replication, 577
write-update, 614
writes, scaling, 483
WriteThrough vs. WriteBack, 418

X

X-25E drives, 404
X.509 certificates, 2
x86 architecture, 390, 431
XA transactions, 314, 330
xdebug, 78
Xeround, 549, 602
xhprof tool, 77
XtraBackup, Percona, 457, 624, 627, 631, 648, 658
XtraDB Cluster, Percona, 516, 549, 577–580, 680

Y

YEAR() function, 268, 270

Z

Zabbix, 668
Zenoss, 669
ZFS filer, 631, 640
ZFS filesystem, 408, 431
zlib, 19, 511
Zmanda Recovery Manager (ZRM), 659

关于作者

Baron Schwartz 是一位软件工程师，居住在弗吉尼亚州的 Charlottesville，网络常用名是 Xaprb，这是按照 QWERTY 键盘的顺序在 Dvorak 键盘上打出来的名字。在不忙于解决有趣的编程挑战时，Baron 会和他的妻子 Lynn 以及小狗 Carbon 一起享受闲暇的时光。他有一个软件工程方面的博客，地址是 *http://www.xaprb.com/blog/*

Peter Zaitsev 曾经是 MySQL AB 公司高性能组的经理，目前在运作 *mysqlperformance blog.com* 网站。他擅长于帮助那些每天有数以百万计访问量的网站的管理员解决问题，这些网站通常需要几百台机器来处理 TB 级的数据。他常常为了解决一个问题而不停地升级硬件和软件（比如查询优化）。Peter 还经常在各种会议上演讲。

Vadim Tkachenko 曾经是 MySQL AB 公司的性能工程师。作为一名在多线程编程和同步方面的专家，他的主要工作是基准测试、性能剖析，以及找出系统的性能瓶颈。他还在性能监控和调优方面做了一些工作，使得 MySQL 在多核机器上有更好的可扩展性。

封面说明

本书封面上的动物是雀鹰（sparrow hawk），又名鹞（Accipiternisus）。它是猎鹰家族的一名成员，生活在欧亚大陆和北非的树林里。雀鹰有着长长的尾巴和短小的翅膀：雄鸟是蓝灰色的，有着浅棕色的胸部；雌鸟大多数是棕灰色的，胸部几乎是纯白色。雄鸟（11 英寸）的体型通常比雌鸟（15 英寸）要小一些。

雀鹰生活在针叶林中，以小型哺乳动物、昆虫和鸟类为食。它们的巢一般筑在树上，有时甚至在悬崖峭壁上。每年夏初，在位于最高的树的主干上的巢里，雌鸟会产下四到六颗白中略带一些红色和棕色的蛋；雄鸟则负责给雌鸟和幼鸟们喂食。

和所有的鹰一样，雀鹰也具有在飞行时高速俯冲的能力。无论是展翅高飞还是盘旋滑翔时，雀鹰都会有带着明显特征的拍翅—拍翅—滑行的动作；它的大尾巴使其能够轻松地盘旋和转身，从容地出入树林之间。

封面图片是一幅 19 世纪的雕版画，出自 Dover Pictorial Archive。

译者简介

宁海元 有超过十年的数据库管理经验，从最初的 SQL Server 2000 到 Oracle 再到 MySQL，擅长数据库高可用架构、性能优化和故障诊断。2007 年加入淘宝，带领淘宝 DBA 团队完成了数据库的垂直拆分、水平拆分，迁移到 MySQL 体系等主要工作，为淘宝业务的快速增长提供支撑。目前专注于无线数据领域。网络常用名 NinGoo，个人博客：*http://www.ningoo.net*

周振兴 毕业于北京师范大学数学系，2009 年加入淘宝数据库团队，负责 MySQL 运维管理工作，有丰富的 MySQL 性能优化、Troubleshooting 经验，对 MySQL 主要模块的实现和原理有深入的研究，经历了淘宝 MySQL 实例从 30 到 3000 的发展，对系统架构、高可用环境规划都有深入理解。个人博客：*http://orczhou.com*

彭立勋 2010 年大学毕业后加入阿里巴巴运维部。作为一名 MySQL DBA，在运维 MySQL 的过程中对 MySQL 和 InnoDB 的一些功能和缺陷进行了补充，编写了多主复制和数据闪回等补丁。目前在阿里集团核心系统研发部数据库组，专注于 MySQL 数据库相关的开发工作。后来一些补丁被 MySQL 之父 Mony 看中，多主复制、线程内存监控等补丁被合并到了 MariaDB 10.0 版本，本人也因此成为 MariaDB 提交组(Maria-captains)成员。

翟卫祥 毕业于武汉大学，研究生阶段从事数据库相关研究。毕业后就职于阿里巴巴集团数据库技术团队至今，主要负责阿里内部 MySQL 代码分支维护，包括 MySQL Bug Fix 及新特性开发。对 MySQL 内核有一定的研究。

刘辉 2008 年毕业于西安电子科技大学计算机系，硕士学位。2011 年加入阿里巴巴集团数据库技术团队，花名希羽，MySQL 内核开发工程师。